

Evelyn Duesterwald (Ed.)

LNCS 2985

Compiler Construction

13th International Conference, CC 2004
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2004
Barcelona, Spain, March/April 2004, Proceedings



Springer

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2985

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

Evelyn Duesterwald (Ed.)

Compiler Construction

13th International Conference, CC 2004
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2004
Barcelona, Spain, March 29 – April 2, 2004
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Evelyn Duesterwald
IBM T.J. Watson Research Center
1101 Kitchawan Rd./Rte 134
Yorktown Heights, NY 10598, USA

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at <<http://dnb.ddb.de>>.

CR Subject Classification (1998): D.3.4, D.3.1, F.4.2, D.2.6, F.3, I.2.2

ISSN 0302-9743

ISBN 3-540-21297-3 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2004
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 10990834 06/3142 5 4 3 2 1 0

Preface

The CC program committee is pleased to present this volume with the proceedings of the 13th International Conference on Compiler Construction (CC 2004). CC continues to provide an exciting forum for researchers, educators, and practitioners to exchange ideas on the latest developments in compiler technology, programming language implementation, and language design. The conference emphasizes practical and experimental work and invites contributions on methods and tools for all aspects of compiler technology and all language paradigms.

This volume serves as the permanent record of the 19 papers accepted for presentation at CC 2004 held in Barcelona, Spain, during April 1–2, 2004. The 19 papers in this volume were selected from 58 submissions. Each paper was assigned to three committee members for review. The program committee met for one day in December 2003 to discuss the papers and the reviews. By the end of the meeting, a consensus emerged to accept the 19 papers presented in this volume. However, there were many other quality submissions that could not be accommodated in the program; hopefully they will be published elsewhere.

The continued success of the CC conference series would not be possible without the help of the CC community. I would like to gratefully acknowledge and thank all of the authors who submitted papers and the many external reviewers who wrote reviews.

I especially thank the program committee members for their efforts and commitment in providing thoughtful and knowledgeable reviews. Special thanks go to our invited keynote speaker, Prof. Mary Lou Soffa from the University of Pittsburgh. Finally, many thanks to the entire ETAPS committee for making CC 2004 possible, especially to Fernando Orejas, the ETAPS 2004 organizing committee chair and to José Luiz Fiadeiro, the current chair of the ETAPS steering committee.

New York, January 2004

Evelyn Duesterwald

Conference Organization

Program Chair

Evelyn Duesterwald, *IBM T.J. Watson Research Center*

Program Committee

Rastislav Bodik, *UC Berkeley, USA*

Evelyn Duesterwald, *IBM T.J. Watson Research Center, USA*

Christine Eisenbeis, *INRIA, France*

Paul Feautrier, *École Normale Supérieure de Lyon, France*

Jeanne Ferrante, *UC San Diego, USA*

Thomas Gross, *ETH Zürich, Switzerland*

Görel Hedin, *Lund University, Sweden*

Michael Hind, *IBM T.J. Watson Research Center, USA*

Nigel Horspool, *University of Victoria, B.C., Canada*

Susan Horwitz, *University of Wisconsin, USA*

Ulrich Kremer, *Rutgers University, USA*

Rainer Leupers, *Technical University of Aachen, Germany*

Josep Llosa, *UPC Barcelona, Spain*

Eduard Mehofer, *University of Vienna, Austria*

Samuel Midkiff, *Purdue University, USA*

Reinhard Wilhelm, *University of Saarbrücken, Germany*

Ben Zorn, *Microsoft, USA*

Referees

G. Almasi	D. Grove	I. Pechtchanski
J. Bauer	T. Harris	T. Proebsting
R. Brown	R. Henriksson	B. Rajkishore
L. Carter	M. Jimenez	F. Reig
P. Clauss	T. Johnson	T. Risset
A. Cohen	M. O'Boyle	S. Scoller
M. Daumas	R. O'Callahan	H. Seidl
B. Decker	J. Knoop	V. Soni
A. Diwan	C. Krantz	S. Thesing
A. Ertl	J. Krinke	X. Vera
L. Fei	K. Lee	C. von Praun
S. Fink	D. Liang	S. Winkel
R. Govindarajan	A. Nilsson	

Table of Contents

Invited Talk

Developing a Foundation for Code Optimization	1
<i>Mary Lou Soffa</i>	

Program Analysis

Analyzing Memory Accesses in x86 Executables	5
<i>Gogul Balakrishnan, Thomas Reps</i>	
The Limits of Alias Analysis for Scalar Optimizations	24
<i>Rezaul A. Chowdhury, Peter Djeu, Brendon Cahoon, James H. Burrill, Kathryn S. McKinley</i>	
Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs	39
<i>Venkatesh Prasad Ranganath, John Hatcliff</i>	
Data Dependence Profiling for Speculative Optimizations	57
<i>Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, Pen-Chung Yew</i>	

Parsing

Elkhound: A Fast, Practical GLR Parser Generator	73
<i>Scott McPeak, George C. Necula</i>	
Generalised Parsing: Some Costs	89
<i>Adrian Johnstone, Elizabeth Scott, Giorgios Economopoulos</i>	

Loop Analysis

An Automata-Theoretic Algorithm for Counting Solutions to Presburger Formulas	104
<i>Erin Parker, Siddhartha Chatterjee</i>	
A Symbolic Approach to Bernstein Expansion for Program Analysis and Optimization	120
<i>Philippe Clauss, Irina Tchoupaeva</i>	

Periodic Polyhedra	134
<i>Benoît Meister</i>	

Optimization

Region-Based Partial Dead Code Elimination on Predicated Code	150
<i>Qiong Cai, Lin Gao, Jingling Xue</i>	
Value-Based Partial Redundancy Elimination	167
<i>Thomas VanDrunen, Antony L. Hosking</i>	
Increasing the Applicability of Scalar Replacement	185
<i>Byoungro So, Mary Hall</i>	
Reducing the Cost of Object Boxing	202
<i>Tim Owen, Des Watson</i>	

Code Generation and Backend Optimizations

FFT Compiler Techniques	217
<i>Stefan Kral, Franz Franchetti, Juergen Lorenz, Christoph W. Ueberhuber, Peter Wurzinger</i>	
Widening Integer Arithmetic	232
<i>Kevin Redwine, Norman Ramsey</i>	
Stochastic Bit-Width Approximation Using Extreme Value Theory for Customizable Processors	250
<i>Emre Özer, Andy P. Nisbet, David Gregg</i>	
Using Multiple Memory Access Instructions for Reducing Code Size	265
<i>Neil Johnson, Alan Mycroft</i>	

Compiler Construction

Integrating the Soot Compiler Infrastructure into an IDE	281
<i>Jennifer Lhoták, Ondřej Lhoták, Laurie Hendren</i>	
Declarative Composition of Stack Frames	298
<i>Christian Lindig, Norman Ramsey</i>	

Author Index	313
-------------------------------	-----

Foreword

ETAPS 2004 was the seventh instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (FOSSACS, FASE, ESOP, CC, TACAS), 23 satellite workshops, 1 tutorial, and 7 invited lectures (not including those that are specific to the satellite events).

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools that support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate program committee and independent proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for “unifying” talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2004 was organized by the LSI Department of the Catalonia Technical University (UPC), in cooperation with:

European Association for Theoretical Computer Science (EATCS)
European Association for Programming Languages and Systems (EAPLS)
European Association of Software Science and Technology (EASST)
ACM SIGACT, SIGSOFT and SIGPLAN

The organizing team comprised

Jordi Cortadella (Satellite Events), Nikos Mylonakis, Robert Nieuwenhuis,
Fernando Orejas (Chair), Edelmira Pasarella, Sonia Perez, Elvira Pino,
Albert Rubio

and had the assistance of TILESA OPC.

ETAPS 2004 received generous sponsorship from:

UPC, Spanish Ministry of Science and Technology (MCYT), Catalan Department for Universities, Research and Information Society (DURSI), IBM, Intel.

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Ratislav Bodik (Berkeley), Maura Cerioli (Genoa), Evelyn Duesterwald (IBM, Yorktown Heights), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Andy Gordon (Microsoft Research, Cambridge), Roberto Gorrieri (Bologna), Nicolas Halbwachs (Grenoble), Görel Hedin (Lund), Kurt Jensen (Aarhus), Paul Klint (Amsterdam), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Hanne Riis Nielson (Copenhagen), Fernando Orejas (Barcelona), Mauro Pezzè (Milan), Andreas Podelski (Saarbrücken), Mooly Sagiv (Tel Aviv), Don Sannella (Edinburgh), Vladimiro Sassone (Sussex), David Schmidt (Kansas), Bernhard Steffen (Dortmund), Perdita Stevens (Edinburgh), Andrzej Tarlecki (Warsaw), Igor Walukiewicz (Bordeaux), Michel Wermelinger (Lisbon)

I would like to express my sincere gratitude to all of these people and organizations, the program committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, and finally Springer-Verlag for agreeing to publish the ETAPS proceedings. This year, the number of submissions approached 600, making acceptance rates fall to 25%. I congratulate the authors who made it into the final program! I hope that all the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

In 2005, ETAPS will be organized by Don Sannella in Edinburgh. You will be welcomed by another “local”: my successor as ETAPS Steering Committee Chair – Perdita Stevens. My wish is that she will enjoy coordinating the next three editions of ETAPS as much as I have. It is not an easy job, in spite of what Don assured me when I succeeded him! But it is definitely a very rewarding one. One cannot help but feel proud of seeing submission and participation records being broken one year after the other, and that the technical program reached the levels of quality that we have been witnessing. At the same time, interacting with the organizers has been a particularly rich experience. Having organized the very first edition of ETAPS in Lisbon in 1998, I knew what they were going through, and I can tell you that each of them put his/her heart, soul, and an incredible amount of effort into the organization. The result, as we all know, was brilliant on all counts! Therefore, my last words are to thank Susanne Graf (2002), Andrzej Tarlecki and Paweł Urzyczyn (2003), and Fernando Orejas (2004) for the privilege of having worked with them.

Developing a Foundation for Code Optimization

Mary Lou Soffa

Department of Computer Science
University of Pittsburgh,
Pittsburgh, PA
soffa@cs.pitt.edu
www.cs.pitt.edu/~soffa

Abstract. Although optimization technology has been successful over the past 40 years, recent trends are emerging that demand we reconsider the paradigm that we are using for code optimization. In particular, the trends toward dynamic optimization, writing embedded system software in high level languages and the lack of significant performance improvement from current optimization research are forcing us to rethink what we know and do not know about optimization. A number of problems dealing with both semantic and application properties of optimizations have always existed but have been mostly ignored. The challenge in optimization today is to explore properties of optimizations and develop a framework for better understanding and use of optimizations. Fortunately, research is starting to explore properties, including proving the soundness of optimizations, proving the correctness of optimizations, specifications of optimizations, and the predictability of profits of optimizations. Not only must we understand the properties, but we also need to integrate the properties into a framework. Only then can we address the problems of the developing trends.

1 Introduction

The field of optimization has been extremely successful over the past 40+ years. As new languages and new architectures have been introduced, new and effective optimizations have been developed to target and exploit both the software and hardware innovations. Various reports from both research and commercial projects have indicated that the performance of software can be improved by 20% to 40% by applying levels of aggressive optimizations.

Most of the success in the field has come from the development of particular optimizations, such as partial redundancy elimination, speculation, and path sensitive optimization. Although we knew that there were various problems with optimization that were not well understood, they were mostly ignored. Thus instead of trying to understand and solve the problems, we avoided them for the most part because we were getting performance improvements. These problems included knowing when, where and what optimizations to apply for the best improvement. Other problems

include showing the soundness of optimizations (an optimization does not change the semantics of a program) and the correctness of the optimizer that implements the optimizations. When optimizations are introduced, seldom is their soundness proved, and likewise optimizers are notorious for being buggy.

2 Technical Challenges

A number of recent events are forcing us to finally take up the challenge of the optimization problems. Because of the continued growth of embedded systems and the competitive market, where time-to-market is critical, there is a movement to write software for embedded system in high level languages. This movement requires an optimizing compiler to generate code that is near the quality of the manually produced code. Today's optimization technology is not able to adequately handle some of the challenges offered by embedded systems. For example, the resource constraints are more severe than in desktop computers and thus optimizations must be able to satisfy the constraints. Furthermore, embedded systems have multiple constraints, including execution time, memory and energy. Most of the prior work in optimization has really focused on a single constraint, namely time. The optimization technology has not been developed to handle the multiple constraints and the trade-offs. Another activity that has brought optimization problems to the forefront is the trend toward dynamic optimization. Dynamic optimization requires that we understand certain properties of optimizations in order for them to be effective. Currently, it is unclear when and where to apply optimizations dynamically and how aggressive the optimizations can be and still be profitable after factoring in the cost of applying the optimization. The third challenge is the lack of performance improvement that we are currently experiencing with optimization research. Although new optimizations continue to be developed, the performance improvement is shrinking. The question then is whether the field has reached its limit or is the problems that we have ignored simply limiting our progress. Lastly, the robustness of software has become a major concern, mandating that we ensure that the optimizing compilers are correct and optimizations sound.

To tackle these problems, we need to better understand the properties of optimization. We categorize optimization properties as either (1) semantic or (2) operational. Semantic properties deal with the semantics of the optimization and include

- correctness – the correctness of the implementation of optimizations,
- soundness – the semantics of a program do not change after applying an optimization, and
- specification – being able to specify conditions both the conditions needed and the semantics of applying the optimization.

Operational properties target the application of optimizations and their performance and include

- interaction – the conditions under which optimizations enable and/or disable other optimizations,
- profitability – the profit of applying an optimization at a particular point in the code given the resources of the targeted platform,
- order – the order that optimizations should be applied, based on the interaction property,
- configuration - the best optimization configuration, including tile size of loop tiling and the unroll factor, considering the resources.
- automatic generation – the conditions under which we can specify optimizations and have a tool that automatically implements the optimizations, and
- portability – the conditions to develop plug-in optimization components.

Research on these properties has been limited. However, more recently there has been a flurry of research activity focusing on optimization properties. There are two approaches to exploring the properties. One approach is through formal techniques. These include developing formal specifications of optimizations, analytical models, and proofs through model checking and theorem provers. Another approach is experimental. That is, the properties are explored by actually implementing optimizations and executing the optimized code, using the information gained to determine properties. The experimental approach can be performed prior to or in conjunction with actual program executions.

3 Related Research

Semantic properties have been formally tackled by proving the soundness of the optimizations [1] [2], and the correctness of the optimizers [3]. Formal specifications of optimizations were introduced a number of years ago by Whitfield and Soffa [4] for automatic implementation of optimizations. Recently, specification techniques have been developed to prove the soundness of optimizations [1] [2]. An experimental approach to the correctness of an optimizer involved checking both unoptimized and optimized code [5].

For operational properties, determining the order or the configuration to apply optimizations has recently been experimentally explored by Triantafyllis et al. [6], Cooper et al. [7], and Kisuki et al. [8]. The interaction and ordering property was formally explored by Whitfield and Soffa [9]. The profitability and predictability of optimizations is also being explored [10] [11].

4 Summary

The important challenge of optimization research today is to better understand properties of optimizations and use these properties when decisions about what optimization to apply, when to apply it and under what conditions to apply it. Also, it

tion to apply, when to apply it and under what conditions to apply it. Also, it is imperative that we know optimizations are sound and that the implementation of optimizations is robust. We need to develop a foundation that can be used to help us determine the properties both for existing optimizations and for future ones, and to integrate the properties. Such a foundation would enable us to better understand optimizations and help us meet the challenges of emerging technologies.

References

- [1] Lacey, E., Jones, N.D., Eric Van Wyk, E., Christian Frederiksen, C., Proving Correctness of Compiler Optimizations by Temporal Logic, Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, (2002) 283-294.
- [2] Lerner, S., Millstein, T., and Chambers, C., Automatically Proving the Correctness of Compiler Optimizations, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, (2003) 1-19.
- [3] Necula, G.C., Translation Validation for an Optimizing Compiler, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, British Columbia, Canada (2000) 83-94.
- [4] Whitfield D., Soffa, M.L., An Approach for Exploring Code Improving Transformations. ACM Transactions on Programming Languages, 19(6) (1997) 1053-1084.
- [5] Jaramillo, C., Gupta, R., Soffa, M.L., Comparison Checking: An approach to avoid debugging of optimized code, ACM SIGSOFT Proceedings of Foundation of Software Engineering, (1999) 268-284.
- [6] Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D., Compiler Optimization-space Exploration. 1st International Symposium on Code Generation and Optimization (2003) 204-215.
- [7] K. Cooper, K., D. Subramanian, D., Torczon, L., Adaptive Optimizing Compilers for the 21st Century, Proceedings of the 2001 LACSI Symposium, Santa Fe, NM, USA, October (2001).
- [8] Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P., Combined Selection of Tile Size and Unroll Factors Using Iterative Compilation, International Conference on Parallel Architectures and Compilation Techniques (2000) 237-246.
- [9] Whitfield, D., Soffa, M.L. An Approach to Ordering Optimizing Transformations, Proceedings ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, (1990) 137-146.
- [10] Zhao, M., Childers, B., Soffa, M.L., Predicting the Impact of Optimizations for Embedded Systems, 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems, San Diego, CA. (2003) 1-11.
- [11] Zhao, M., Childers, B., Soffa, M.L., Profit Driven Optimizations, Technical Report, University of Pittsburgh, Jan. (2004).

Analyzing Memory Accesses in x86 Executables *

Gogul Balakrishnan and Thomas Reps

Comp. Sci. Dept., University of Wisconsin;
`{bgogul, reps}@cs.wisc.edu`

Abstract. This paper concerns static-analysis algorithms for analyzing x86 executables. The aim of the work is to recover intermediate representations that are similar to those that can be created for a program written in a high-level language. Our goal is to perform this task for programs such as plugins, mobile code, worms, and virus-infected code. For such programs, symbol-table and debugging information is either entirely absent, or cannot be relied upon if present; hence, the technique described in the paper makes no use of symbol-table/debugging information. Instead, an analysis is carried out to recover information about the contents of memory locations and how they are manipulated by the executable.

1 Introduction

In recent years, there has been a growing need for tools that analyze executables. One would like to ensure that web-plugins, Java applets, etc., do not perform any malicious operations, and it is important to be able to decipher the behavior of worms and virus-infected code. Static analysis provides techniques that can help with such problems. A major stumbling block when developing binary-analysis tools is that it is difficult to understand memory operations because machine-language instructions use explicit memory addresses and indirect addressing. In this paper, we present several techniques that overcome this obstacle to developing binary-analysis tools.

Just as source-code-analysis tools provide information about the contents of a program's variables and how variables are manipulated, a binary-analysis tool should provide information about the contents of memory locations and how they are manipulated. Existing techniques either treat memory accesses extremely conservatively [4,6,2], or assume the presence of symbol-table or debugging information [27]. Neither approach is satisfactory: the former produces very approximate results; the latter uses information that cannot be relied upon when analyzing viruses, worms, mobile code, etc. Our analysis algorithm can do a better job than previous work because it tracks the pointer-valued and integer-valued quantities that a program's data objects can hold, using a set of abstract data objects, called *a-locs* (for “abstract locations”). In particular, the analysis is not forced to give up all precision when a load from memory is encountered.

The idea behind the a-loc abstraction is to exploit the fact that accesses on the variables of a program written in a high-level language appear as either static addresses (for globals) or static stack-frame offsets (for locals). Consequently, we find all the

* Supported by ONR contracts N00014-01-1-{0708,0796} and NSF grant CCR-9986308.

statically known locations and stack offsets in the program, and define an a-loc to be the set of locations from one statically known location/offset up to, but not including the next statically known location/offset. (The registers and `malloc` sites are also a-locs.) As discussed in §3.2, the data object in the original source-code program that corresponds to a given a-loc can be one or more scalar, struct, or array variables, but can also consist of just a segment of a scalar, struct, or array variable.

Another problem that arises in analyzing executables is the use of indirect-addressing mode for memory operands. Machine-language instruction sets normally support two addressing modes for memory operands: direct and indirect. In direct addressing, the address is in the instruction itself; no analysis is required to determine the memory location (and hence the corresponding a-loc) referred to by the operand. On the other hand, if the instruction uses indirect addressing, the address is typically specified through a register expression of the form $\text{base} + \text{index} \times \text{scale} + \text{offset}$ (where `base` and `index` are registers). In such cases, to determine the memory locations referred to by the operand, the values that the registers hold at this instruction need to be determined. We present a flow-sensitive, context-insensitive analysis that, for each instruction, determines an over-approximation to the set of values that each a-loc could hold.

The contributions of our work can be summarized as follows:

- We describe a static-analysis algorithm, *value-set analysis*, for tracking the values of data objects (other than just the hardware registers). Value-set analysis uses an abstract domain for representing an over-approximation of the set of values that each data object can hold at each program point. The algorithm tracks address-valued and integer-valued quantities simultaneously: it determines an over-approximation of the set of addresses that each data object can hold at each program point; at the same time, it determines an over-approximation of the set of integer values that each data object can hold at each program point.
- Value-set analysis can be used to obtain used, killed, and possibly-killed sets for each instruction in the program. These sets are similar to the sets of used, killed, and possibly-killed variables obtained by a compiler in some source-code analyses. They can be used to perform reaching-definitions analysis and to construct data-dependence edges.
- We have implemented the analysis techniques described in the paper. By combining this analysis with facilities provided by the IDAPro [17] and CodeSurfer® [7] toolkits, we have created CodeSurfer/x86, a prototype tool for browsing, inspecting, and analyzing x86 executables. This tool recovers IRs from x86 executables that are similar to those that can be created for a program written in a high-level language. The paper reports preliminary performance data for this implementation.

The information obtained from value-set analysis should also be useful in decompilation tools. Although the implementation is targeted for x86 executables, the techniques described in the paper should be applicable to other machine languages.

Some of the benefits of our approach are illustrated by the following example:

Example 1. Fig. 1 shows a simple C program and the corresponding disassembly. Procedure `main` declares an integer array `a` of ten elements. The program initializes the first five elements of `a` with the value of `part1Value`, and the remaining five with `part2Value`. It then returns `*p_array0`, i.e., the first element of `a`.

A diagram of how variables are laid out in the program’s address space is shown in Fig. 2(a). To understand the assembly program in Fig. 1, it helps to know that

```

int part1Value=0;
int part2Value=1;

int main() {
    int *part1,*part2;
    int a[10],*p_array0;
    int i;
    part1=&a[0];
    p_array0=part1;
    part2=&a[5];
    for(i=0;i<5;++i) {
        *part1=part1Value;
        *part2=part2Value;
        part1++;
        part2++;
    }
    return *p_array0;
}

proc main          ;
1  sub esp, 44      ;Adjust esp for locals
2  lea eax, [esp+4] ;part1=&a[0]
3  lea ebx, [esp+24];part2=&a[5]
4  mov [esp+0], eax;p_array0=part1
5  mov ecx, 0       ;i=0
L1: mov edx, [4]    ;
7   mov [eax], edx  ;*part1=part1Value
8   mov edx, [8]    ;
9   mov [ebx], edx  ;*part2=part2Value
10  add eax, 4     ;part1++
11  add ebx, 4     ;part2++
12  inc ecx        ;i++
13  cmp ecx, 5     ;
14  jl L1          ;(i<5)?loop:exit
15  mov edi, [esp+0] ;
16  mov eax, [edi]  ;set return value
17  add esp, 44     ;
18  retn            ;return *p_array0

```

Fig. 1. A C program that initializes an array.

- The address of global variable `part1Value` is 4 and that of `part2Value` is 8.
- The local variables `part1`, `part2`, and `i` of the C program have been removed by the optimizer and are mapped to registers `eax`, `ebx`, and `ecx`.
- The instruction that modifies the first five elements of the array is “7: `mov [eax], edx`”; the one that modifies the last five elements is “9: `mov [ebx], edx`”.

The statements that are underlined in Fig. 1 show the backward slice of the program with respect to `16 mov eax, [edi]`—which roughly corresponds to `return (*p_array0)` in the source code—that would be obtained using the sets of used, killed, and possibly-killed a-locs identified by value-set analysis. The slice obtained with this approach is actually smaller than the slice obtained by most source-code slicing tools. For instance, CodeSurfer/C does not distinguish accesses to different parts of an array. Hence, the slice obtained by CodeSurfer/C from C source code would include all of the statements in Fig. 1, not just the underlined ones. \square

The following insights shaped the design of value-set analysis:

- To prevent most indirect-addressing operations from appearing to be possible non-aligned accesses that span parts of two variables—and hence possibly forging new pointer values—it is important for the analysis to discover information about the alignments and strides of memory accesses.
- To prevent most loops that traverse arrays from appearing to be possible stack-smashing attacks, the analysis needs to use relational information so that the values of a-locs assigned to within a loop can be related to the values of the a-locs used in the loop’s branch condition.
- It is desirable for the analysis to perform pointer analysis and numeric analysis simultaneously: information about numeric values can lead to improved tracking of

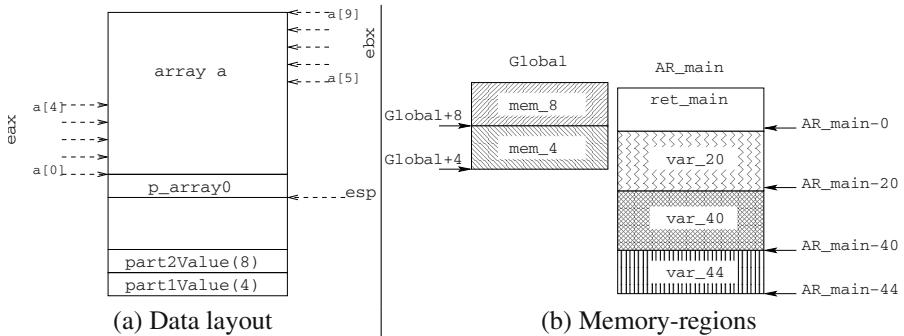


Fig. 2. Data layout and memory-regions for Example 1.

pointers, and pointer information can lead to improved tracking of numeric values. This appears to be a crucial capability, because compilers use address arithmetic and indirect addressing to implement such features as pointer arithmetic, pointer dereferencing, array indexing, and accessing structure fields.

Value-set analysis produces information that is more precise than that obtained via several more conventional numeric analyses used in compilers, including constant propagation, range analysis, and integer-congruence analysis. At the same time, value-set analysis provides an analog of pointer analysis that is suitable for use on executables.

Debray et al. [11] proposed a flow-sensitive, context-insensitive algorithm for analyzing an executable to determine if two address expressions may be aliases. Our analysis yields more precise results than theirs: for the program shown in Fig. 1, their algorithm would be unable to determine the value of `edi`, and so the analysis would consider `[edi]`, `[eax]`, and `[ebx]` to be aliases of each other. Hence, the slice obtained using their alias analysis would also consist of the whole program. Cifuentes et al. [5] proposed a static-slicing algorithm for executables. They only consider programs with non-aliased memory locations, and hence would identify an unsafe slice of the program in Fig. 1, consisting only of the instructions 16, 15, 4, 2, and 1. (See §9 for a more detailed discussion of related work.)

The remainder of the paper is organized as follows: §2 describes how value-set analysis fits in with the other components of CodeSurfer/x86, and discusses the assumptions that underlie our work. §3 describes the abstract domain used for value-set analysis. §4 describes the value-set analysis algorithm. §5 summarizes an auxiliary static analysis whose results are used during value-set analysis when interpreting conditions and when performing widening. §6 discusses indirect jumps and indirect function calls. §7 presents preliminary performance results. §8 discusses soundness issues. §9 discusses related work. (Value-set analysis will henceforth be referred to as VSA.)

2 The Context of the Problem

CodeSurfer/x86 is the outcome of a joint project between the Univ. of Wisconsin and GrammaTech, Inc. CodeSurfer/x86 makes use of both IDAPro [17], a disassembly toolkit, and GrammaTech’s CodeSurfer system [7], a toolkit for building

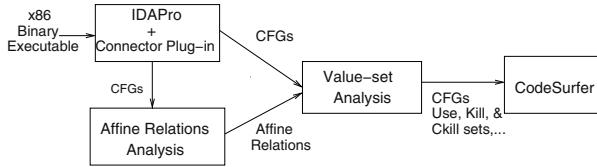


Fig. 3. Organization of CodeSurfer/x86.

program-analysis and inspection tools. This section describes how VSA fits into the CodeSurfer/x86 implementation.

The x86 executable is first disassembled using IDAPro. In addition to the disassembly listing, IDAPro also provides access to the following information:

Statically known memory addresses and offsets: IDAPro identifies the statically known memory addresses and stack offsets in the program, and renames all occurrences of these quantities with a consistent name. We use this database to define the a-locs.

Information about procedure boundaries: X86 executables do not have information about procedure boundaries. IDAPro identifies the boundaries of most of the procedures in an executable.¹

Calls to library functions: IDAPro discovers calls to library functions using an algorithm called the Fast Library Identification and Recognition Technology (FLIRT) [13]. This information is necessary to identify calls to `malloc`.

IDAPro provides access to its internal data structures via an API that allows users to create plug-ins to be executed by IDAPro. GrammaTech provided us with a plugin to IDAPro (called the Connector) that augments IDAPro's data structures. VSA is implemented using the data structures created by the Connector. As described in §5, VSA makes use of the results of an additional preliminary analysis, which, for each program point, identifies the affine relations that hold among the values of registers. Once VSA completes, the value-sets for the a-locs at each program point are used to determine each point's sets of used, killed, and possibly-killed a-locs; these are emitted in a format that is suitable for input to CodeSurfer.

CodeSurfer is a tool for code understanding and code inspection that supports both a GUI and an API for accessing a program's system dependence graph (SDG) [16], as well as other information stored in CodeSurfer's intermediate representations (IRs). CodeSurfer's GUI supports browsing ("surfing") of an SDG, along with a variety of operations for making queries about the SDG—such as slicing [16] and chopping [25]. The API can be used to extend CodeSurfer's capabilities by writing programs that traverse CodeSurfer's IRs to perform additional program analyses.

A few words are in order about the goals, capabilities, and assumptions underlying our work:

¹ IDAPro does not identify the targets of all indirect jumps and indirect calls, and therefore the call graph and control-flow graphs that it constructs are not complete. §6 discusses techniques for using the abstract stores computed during VSA to augment the call graph and control-flow graphs on-the-fly to account for indirect jumps and indirect calls.

- Given an executable as input, the goal is to check whether the executable conforms to a “standard” compilation model—i.e., a runtime stack is maintained; activation records (ARs) are pushed on procedure entry and popped on procedure exit; each global variable resides at a fixed offset in memory; each local variable of a procedure f reside at a fixed offset in the ARs for f ; actual parameters of f are pushed onto the stack by the caller so that the corresponding formal parameters reside at fixed offsets in the ARs for f ; the program’s instructions occupy a fixed area of memory, are not self-modifying, and are separate from the program’s data.

If the executable does conform to this model, the system will create an IR for it. If it does not conform, then one or more violations will be discovered, and corresponding error reports will be issued (see §8).

We envision CodeSurfer/x86 as providing (i) a tool for security analysis, and (ii) a general infrastructure for additional analysis of executables. Thus, in practice, when the system produces an error report, a choice is made about how to accommodate the error so that analysis can continue (i.e., the error is optimistically treated as a false positive), and an IR is produced; if the user can determine that the error report is indeed a false positive, then the IR is valid.

- The analyzer does not care whether the program was compiled from a high-level language, or hand-written in assembly. In fact, some pieces of the program may be the output from a compiler (or from multiple compilers, for different high-level languages), and others hand-written assembly.
- In terms of what features a high-level-language program is permitted to use, VSA is capable of recovering information from programs that use global variables, local variables, pointers, structures, arrays, heap-allocated storage, pointer arithmetic, indirect jumps, recursive procedures, and indirect calls through function pointers (but not runtime code generation or self-modifying code).
- Compiler optimizations often make VSA *less* difficult, because more of the computation’s critical data resides in registers, rather than in memory; register operations are more easily deciphered than memory operations.
- The major assumption that we make is that IDAPro is able to disassemble a program and build an adequate collection of *preliminary* IRs for it. Even though (i) the CFG created by IDAPro may be incomplete due to indirect jumps, and (ii) the call-graph created by IDAPro may be incomplete due to indirect calls, incomplete IRs do *not* trigger error reports. Both the CFG and the call-graph will be fleshed out according to information recovered during the course of VSA (see §6). In fact, the relationship between VSA and the preliminary IRs created by IDAPro is similar to the relationship between a points-to-analysis algorithm in a C compiler and the preliminary IRs created by the C compiler’s front end. In both cases, the preliminary IRs are fleshed out during the course of analysis.

3 The Abstract Domain

The abstract stores used during VSA over-approximate sets of concrete stores. Abstract stores are based on the concepts of *memory-regions* and *a-locs*, which are discussed first.

3.1 Memory-Regions

Memory addresses in an executable for an x -bit machine are x -bit numbers. Hence, one possible approach would be to use an existing numeric static-analysis domain, such as

intervals [8], congruences [14], etc., to over-approximate the set of values (including addresses) that each data object can hold. However, there are several problems with such an approach: (1) addresses get reused, i.e., the same address can refer to different program variables at runtime; (2) a variable can have several runtime addresses; and (3) addresses cannot be determined statically in certain cases (e.g., memory blocks allocated from the heap via `malloc`).

Even though the same address can be shared by multiple ARs, it is possible to distinguish among these addresses based on what procedure is active at the time the address is generated (i.e., a reference to a local variable of `f` does not refer to a local variable of `g`). VSA uses an analysis-time analog of this: We assume that the address-space of a process consists of several non-overlapping regions called *memory-regions*. For a given executable, the set of memory-regions consists of one region per procedure, one region per heap-allocation statement, and a global region. We do not assume anything about the relative positions of these memory-regions. The region associated with a procedure represents all instances of the procedure’s runtime-AR. Similarly, the region associated with a heap-allocation statement represents all memory blocks allocated by that statement at runtime. The global region represents the uninitialized-data and initialized-data sections of the program.

Fig. 2(b) shows the memory-regions for the program from Fig. 1. There is a single procedure, and hence two regions: one for global data and one for the AR of `main`.

The analysis treats all data objects, whether local, global, or in the heap, in a fashion similar to the way compilers arrange to access variables in local ARs, namely, via an offset. We adopt this notion as part of our concrete semantics: a “concrete” memory address is represented by a pair: (memory-region, offset). (Thus, the concrete semantics already has a degree of abstraction built into it.) As explained below, an abstract memory address will track possible offsets using a numeric abstraction.

For the program from Fig. 1, the address of local variable `p_array0` is the pair (`AR_main`, -44), and that of global variable `part2Value` is (`Global`, 8).

At the enter node of a procedure `P`, register `esp` points to the start of the AR of `P`. Therefore, the enter node of a procedure `P` is considered to be a statement that initializes `esp` with the address (`AR_P`, 0). A call on `malloc` at program point `L` is considered to be a statement that assigns the address (`malloc_L`, 0).

3.2 A-Locs

Indirect addressing in x86 instructions involves only registers. However, it is not sufficient to track values only for registers, because registers can be loaded with values from memory. If the analysis does not also track an approximation of the values that memory locations can hold, then memory operations would have to be treated conservatively, which would lead to very imprecise data dependences. Instead, we use what we call the *a-loc* abstraction to track (an over-approximation of) the values of memory locations.

An a-loc is roughly equivalent to a variable in a C program. The a-loc abstraction is based on the following observation: the data layout of the program is established before generating the executable, i.e., the compiler or the assembly-programmer decides where to place the global variables, local variables, etc. Globals will be accessed via direct operands in the executable. Similarly, locals will be accessed via indirect operands with `esp` (or `ebp`) as the base register, but a constant offset. Thus, examination of direct and indirect operands provides a rough idea of the base addresses and sizes of the program’s

variables. Consequently, we define an a-loc to be the set of locations between two such consecutive addresses or offsets.

For the program from Fig. 1, the direct operands are [4] and [8]. Therefore, we have two a-locs: `mem_4` (for addresses 4..7) and `mem_8` (for addresses 8..11). Also, the `esp/ebp`-based indirect operands are `[esp+0]`, `[esp+4]`, and `[esp+24]`. These operands are accesses on the local variables in the AR of `main`. On entry to `main`, `esp = (AR_main, 0)`; the difference between the value of `esp` on entry to `main` and the value of `esp` at these operands is -44 . Thus, these memory references correspond to the offsets -44 , -40 , and -20 in the memory-region for `AR_main`. This gives rise to three more a-locs: `var_44`, `var_40`, and `var_20`. In addition to these a-locs, an a-loc for the return address is also defined; its offset in `AR_main` is 0.

Note that `var_44` corresponds to all of the source-code variable `p_array0`. In contrast, `var_40` and `var_20` correspond to disjoint segments of array `a[]`: `var_40` corresponds to `a[0..4]`; `var_20` corresponds to `a[5..9]`.

Similarly, we have one a-loc per heap-region. In addition to these a-locs, registers are also considered to be a-locs.

Offsets of an a-loc: Once the a-locs are identified, the relative positions of these a-locs in their respective regions are also recorded. The offset of an a-loc a in a region rgn will be denoted by $\text{offset}(rgn, a)$. For example, for the program from Fig. 1, $\text{offset}(\text{AR_main}, \text{var_20})$ is -20 .

Addresses of an a-loc: The addresses that belong to an a-loc a can be represented by a pair $(rgn, [offset, offset + size - 1])$, where rgn represents the memory region to which it belongs to, $offset$ is the offset of the a-loc within the region, and $size$ is the size of the a-loc. A pair of the form $[a, b]$ represents the set of integers $\{x | a \leq x \leq b\}$. For the program from Fig. 1, the addresses of a-loc `var_20` are $(\text{AR_main}, [-40, -21])$. The $size$ of an a-loc may not be known for heap a-locs. In such cases, $size = \infty$.

3.3 Abstract Stores

An abstract store must over-approximate the set of memory addresses that each a-loc holds at a particular program point. As described in §3.1, every memory address is a pair (memory-region, offset). Therefore, a set of memory addresses in a memory region rgn is represented as $(rgn, \{o_1, o_2, \dots, o_n\})$. The offsets o_1, o_2, \dots, o_n are numbers; they can be represented (i.e., over-approximated) using a numeric abstract domain, such as intervals, congruences, etc. We use a reduced interval congruence (RIC) for this purpose. A reduced interval congruence is the reduced cardinal product [9] of an interval domain and a congruence domain. For example, the set of numbers $\{1, 3, 5, 9\}$ can be over-approximated as the RIC $(2\mathbb{Z} + 1) \cap [0, 9]$. Each RIC can be represented as a 4-tuple: the tuple (a, b, c, d) stands for $a \times [b, c] + d$, and denotes the set of integers $\{aZ + d | Z \in [b, c]\}$.² For instance, $\{1, 3, 5, 9\}$ is over-approximated by the tuple $(2, 0, 4, 1)$, which equals $\{1, 3, 5, 7, 9\}$.

An *abstract store* is a value of type $\text{a-loc} \rightarrow (\text{memory-region} \rightarrow \text{RIC})$. For conciseness, the abstract stores that represent addresses in an a-loc for different memory-regions will be combined together into an r -tuple of RICs, where r is the number of memory regions. Such an r -tuple will be referred to as a *value-set*. Thus, an abstract store is a map from a-locs to value-sets: $\text{a-loc} \rightarrow \text{RIC}^r$. For instance, for the program from Fig. 1,

² Because b is allowed to have the value $-\infty$, we cannot always adjust c and d so that b is 0.

at statement 7, `eax` holds the addresses of the first five elements of `main`'s local array, and thus the abstract store maps `eax` to the value-set $(\perp, 4[0, 4] - 40)$.

We chose to use RICs because, in our context, it is important for the analysis to discover alignment and stride information so that it can interpret indirect-addressing operations that implement either (i) field-access operations in an array of structs, or (ii) pointer-dereferencing operations.

When the contents of an a-loc `a` is not aligned with the boundaries of a-locs, a memory access on `*a` can fetch portions of two a-locs; similarly, a write to `*a` can overwrite portions of two a-locs. Such operations can be used to forge new addresses. For instance, suppose that the address of a-loc `x` is 1000, the address of a-loc `y` is 1004, and the contents of `a` is 1001. Then `*a` (as a 4-byte fetch) would retrieve 3 bytes of `x` and 1 byte of `y`.

This issue motivated the use of RICs because RICs are capable of representing certain non-convex sets of integers, and ranges (alone) are not. Suppose that the contents set of `a` is $\{1000, 1004\}$; then `*a` (as a 4-byte fetch) would retrieve `x` or `y`. The range $[1000, 1004]$ includes the addresses 1001, 1002, and 1003, and hence $*[1000, 1004]$ (as a 4-byte fetch) could result in a forged address. However, because VSA is based on RICs, $\{1000, 1004\}$ is represented exactly, as the RIC $4[0,1]+1000$. If VSA were based on range information rather than RICs, it would either have to try to track *segments* of (possible) contents of data objects, or treat such dereferences conservatively by returning \top , thereby losing track of all information.

Value-sets form a lattice. The following operators are defined for value-sets. All operators are pointwise applications of the corresponding RIC operator.

- $(vs_1 \sqsubseteq vs_2)$: Returns true if the value-set vs_1 is a subset of vs_2 , false otherwise.
- $(vs_1 \sqcap vs_2)$: Returns the intersection (meet) of value-sets vs_1 and vs_2 .
- $(vs_1 \sqcup vs_2)$: Returns the union (join) of value-sets vs_1 and vs_2 .
- $(vs_1 \nabla vs_2)$: Returns the value-set obtained by widening vs_1 with respect to vs_2 , e.g., if $vs_1 = (10, 4[0, 1])$ and $vs_2 = (10, 4[0, 2])$, then $(vs_1 \nabla vs_2) = (10, 4[0, \infty])$.
- $(vs \boxplus c)$: Returns the value-set obtained by adjusting all values in vs by the constant c , e.g., if $vs = (4, 4[0, 2] + 4)$ and $c = 12$, then $(vs \boxplus c) = (16, 4[0, 2] + 16)$.
- $*(vs, s)$: Returns a pair of sets (F, P) . F represents the set of “fully accessed” a-locs: it consists of the a-locs that are of size s and whose starting addresses are in vs . P represents the set of “partially accessed” a-locs: it consists of (i) a-locs whose starting addresses are in vs but are not of size s , and (ii) a-locs whose addresses are in vs but whose starting addresses and sizes do not meet the conditions to be in F .
- $\text{RemoveLowerBounds}(vs)$: Returns the value-set obtained by setting the lower bound of each component RIC to $-\infty$. For example, if $vs = ([0, 100], [100, 200])$, then $\text{RemoveLowerBounds}(vs) = ([-\infty, 100], [-\infty, 200])$.
- $\text{RemoveUpperBounds}(vs)$: Similar to RemoveLowerBounds , but sets the upper bound of each component to ∞ .

To represent the abstract store at each program point efficiently, we use applicative dictionaries, which provide a space-efficient representation of a collection of dictionary values when many of the dictionary values have nearly the same contents as other dictionary values in the collection [26,21].

4 Value-Set Analysis (VSA)

This section describes the value-set analysis algorithm. VSA is an abstract interpretation of the executable to find a safe approximation for the set of values that each data object holds at each program point. It uses the domain of abstract stores defined in §3. The present implementation of VSA is flow-sensitive and context-insensitive.³

VSA has similarities with the pointer-analysis problem that has been studied in great detail for programs written in high-level languages. For each variable (say v), pointer analysis determines an over-approximation of the set of variables whose addresses v can hold. Similarly, VSA determines an over-approximation of the set of addresses that each data object can hold at each program point. The results of VSA can also be used to find the a-locs whose addresses a given a-loc a contains. On the other hand, VSA also has some of the flavor of numeric static analyses, where the goal is to over-approximate the integer values that each variable can hold. In addition to information about addresses, VSA determines an over-approximation of the set of integer values that each data object can hold at each program point.

4.1 Intraprocedural Analysis

This subsection describes an intraprocedural version of VSA. For the time being, we will consider programs that have a single procedure and no indirect jumps. To aid in explaining the algorithm, we adopt a C-like notation for program statements. We will discuss the following kinds of instructions, where $R1$ and $R2$ are two registers of the same size, and c , c_1 , and c_2 are explicit integer constants:

$$\begin{array}{ll} R1 = R2 + c & R1 \leq c \\ *(R1 + c_1) = R2 + c_2 & R1 \geq R2 \\ R1 = *(R2 + c_1) + c_2 & \end{array}$$

Conditions of the last two forms are obtained from the predecessor(s) of conditional jump instructions that affect condition codes.

The analysis is performed on a CFG for the procedure. The CFG consists of one node per x86 instruction; the edges are labeled with the instruction at the source of the edge. If the source of an edge is a conditional, then the edge is labeled according to the outcome of the conditional. For instance, the edge $14 \rightarrow L1$ will be labeled $\text{ecx} < 5$, whereas the edge $14 \rightarrow 15$ will be labeled $\text{ecx} \geq 5$. Once we have the CFG, an abstract store is obtained for each program point by abstract interpretation [8]. Sample transformers for various kinds of edges are listed in Fig. 4. Each transformer takes an abstract store and returns a new abstract store. Because each AR region of a procedure that may be called recursively—as well as each heap region—potentially represents more than one concrete data object, assignments to their a-locs must be modeled by weak updates, i.e., the new value-set must be unioned with the existing one, rather than replacing it (see case two of Fig. 4). Furthermore, unaligned writes can modify parts of various a-locs (which could possibly create forged addresses). In case 2 of Fig. 4, such writes are treated safely by setting the values of all partially modified a-locs to \top . Similarly, case 3 treats a load of a potentially forged address as a load of \top .

³ In the near future, we plan to extend the implementation to have a degree of context-sensitivity, using the call-strings approach to interprocedural dataflow analysis [29].

Label on e	Transfer function for edge e
R1=R2+c	let $(R2 \mapsto vs) \in e.\text{Before}$ $e.\text{After} := e.\text{Before} - [R1 \mapsto *] \cup [R1 \mapsto vs \boxplus c]$
$*(R1+c_1)=R2+c_2$	let $[R1 \mapsto vs_{R1}], [R2 \mapsto vs_{R2}] \in e.\text{Before}, (F, P) = *(vs_{R1} \boxplus c_1, s)$, tmp = $e.\text{Before} - \{[a \mapsto *] \mid a \in P \cup F\} \cup \{[p \mapsto \top] \mid p \in P\}$, and Proc be the procedure containing the statement if $(F =1 \text{ and } P =0 \text{ and }$ $(\text{Proc is not recursive}) \text{ and } (F \text{ has no heap objects})$) then $e.\text{After} := (\text{tmp} \cup \{[v \mapsto vs_{R2} \boxplus c_2] \mid v \in F\}) // \text{ Strong update}$ else // Weak update $e.\text{After} := (\text{tmp} \cup \{[v \mapsto (vs_{R2} \boxplus c_2) \sqcup vs_v] \mid v \in F, [v \mapsto vs_v] \in e.\text{Before}\})$
$R1=*(R2+c_1)+c_2$	let $(R2 \mapsto vs_{R2}) \in e.\text{Before}$ and $(F, P) = *(vs_{R2} \boxplus c_1, s)$ if $ P =0$ then let $vs_{rhs} = \bigcup \{[vs_v] \mid v \in F, [v \mapsto vs_v] \in e.\text{Before}\}$ $e.\text{After} := e.\text{Before} - [R1 \mapsto *] \cup [R1 \mapsto (vs_{rhs} \boxplus c_2)]$ else $e.\text{After} := e.\text{Before} - [R1 \mapsto *] \cup [\top \mapsto \top]$
$R1 \leq c$	let $[R1 \mapsto vs_{R1}] \in e.\text{Before}$ and $vs_c = ([-\infty, c], \top, \dots, \top)$ $e.\text{After} := e.\text{Before} - [R1 \mapsto *] \cup [R1 \mapsto vs_{R1} \sqcap vs_c]$
$R1 \geq R2$	let $[R1 \mapsto vs_{R1}], [R2 \mapsto vs_{R2}] \in e.\text{Before}$ and $vs_{lb} = \text{RemoveUpperBounds}(vs_{R2})$ $e.\text{After} := e.\text{Before} - [R1 \mapsto *] \cup [R1 \mapsto vs_{R1} \sqcap vs_{lb}]$

Fig. 4. Transfer functions for VSA. (In cases 2 and 3, s represents the size of the dereference performed by the instruction.)

The abstract store for the entry node consists of the information about the initialized global variables and the initial value of the stack pointer (esp).

The abstract domain has infinite ascending chains. Hence, to ensure termination, widening needs to be performed. Widening needs to be carried out at least one node of every cycle in the CFG; however, the node at which widening is performed can affect the accuracy of the analysis. To choose widening points, our implementation of VSA uses techniques from [3].

Example 2. For the program from Fig. 1, the abstract store for the entry node of main is $\{\text{esp} \mapsto (\perp, 0), \text{mem_4} \mapsto (0, \perp), \text{mem_8} \mapsto (1, \perp)\}$.

The fixpoint solution of VSA for instruction 7 is $\{\text{esp} \mapsto (\perp, -44), \text{mem_4} \mapsto (0, \perp), \text{mem_8} \mapsto (1, \perp), \text{eax} \mapsto (\perp, 4[0, \infty] - 40), \text{ebx} \mapsto (\perp, 4[0, \infty] - 20), \text{var_44} \mapsto (\perp, -40), \text{ecx} \mapsto ([0, 4], \perp)\}$ and that of instruction 16 is $\{\text{esp} \mapsto (\perp, -44), \text{mem_4} \mapsto (0, \perp), \text{mem_8} \mapsto (1, \perp), \text{eax} \mapsto (\perp, 4[1, \infty] - 40), \text{ebx} \mapsto (\perp, 4[1, \infty] - 20), \text{var_44} \mapsto (\perp, -40), \text{ecx} \mapsto ([5, 5], \perp), \text{edi} \mapsto (\perp, -40)\}$.

Note that the value-sets obtained by the analysis can be used to discover the data dependence that exists between instructions 7 and 16. At instruction 7, eax $\mapsto (\perp, 4[0, \infty] - 40)$, and thus $*(\text{eax} \boxplus 0, 4)$ returns the possibly-killed set as $\{\text{var_40}, \text{var_20}, \text{ret_main}\}$. Similarly, at instruction 16, $*(\text{esp} \boxplus 8, 4)$ returns the use set as $\{\text{var_40}\}$. Reaching-definitions analysis based on this information reveals that instruction 16 is data dependent on instruction 7. Similarly, reaching-definitions analysis reveals that instruction 16 is not data dependent on 9.

Note that the a-loc `ret_main` is also included in the set of a-locs accessed through eax at instruction 7. This is because the analysis was not able to determine an upper bound for eax. Observe that eax is dependent on the loop variable ecx. We discuss in §5 how the implemented system actually finds upper or lower bounds for variables that are dependent on the loop variable. \square

4.2 Interprocedural Analysis

Let us now consider procedure calls, but for now ignore indirect jumps and calls. Interprocedural analysis presents new problems because the formals of a procedure and the actuals of a call need to be identified. This information is not directly available in the disassembly because parameters are typically passed on the stack in the x86 architecture. Further, the instructions that push the actual parameters on the stack need not occur immediately before the call. Example 3 will be used to explain the interprocedural case.

Example 3. Fig. 5 shows a program with two procedures, `main` and `initArray` (see also Fig. 6). Procedure `main` has an integer array `a`, which is initialized by calling `initArray`. After initialization, `main` returns the second element of array `a`. \square

Actual Parameters and Register Saves

In an x86 program, stack operations like `push/pop` implicitly modify some locations in the AR of a procedure (say P). These locations correspond to the actual parameters of a call and to those used for register spilling and caller-saved registers. The locations accessed by `push/pop` instructions are not explicitly found as `esp/ebp-relative` addresses, and so the algorithm that identifies a-locs will not introduce a-locs for the memory locations accessed by these stack operations; consequently, we introduce additional a-locs, which we call *extended a-locs*, for memory locations that are implicitly accessed by such stack operations. To do this, the smallest `sp_delta` for P is determined. This represents the maximum limit to which the stack can grow in a single invocation of P. (The stack can grow deeper due to calls made by P; however, these operations are not relevant because we are concerned merely with identifying the size of the AR for P.) If we are unable to find a finite minimum, the analysis issues a report. If there is a finite minimum, then *extended a-locs* are added to the AR on 4-byte boundaries to fill the space between the lowest local a-loc and the minimum `sp_delta`.

```

proc initArray
1 lea eax, [esp+4]
2 mov ebx, eax
3 add ebx, 20
4 mov ecx, 0
L1: mov edx, [4]
5 mov [eax], edx
6 mov edx, [8]
7 mov edx, [8]
8 mov [ebx], edx
9 add eax, 4
10 add ebx, 4
11 inc ecx
12 cmp ecx, [esp+8]
13 jl L1
14 ret

proc main
15 sub esp, 44
16 lea eax, [esp+4]
17 mov [esp+0], eax
18 push 5
19 push eax
20 call initArray
21 add esp, 8
22 mov edi, [esp+0]
23 mov eax, [edi]
24 add esp, 44
25 ret
}

int main(){
    int i=[10], *p.array0;
    p.array0=&a[0];
    initArray(a, 5);
    return *p.array0;
}

```

(a) C program

```

proc initArray
1 lea eax, [esp+4]
2 mov ebx, eax
3 add ebx, 20
4 mov ecx, 0
L1: mov edx, [4]
5 mov [eax], edx
6 mov edx, [8]
7 mov edx, [8]
8 mov [ebx], edx
9 add eax, 4
10 add ebx, 4
11 inc ecx
12 cmp ecx, [esp+8]
13 jl L1
14 ret

proc main
15 sub esp, 44
16 lea eax, [esp+4]
17 mov [esp+0], eax
18 push 5
19 push eax
20 call initArray
21 add esp, 8
22 mov edi, [esp+0]
23 mov eax, [edi]
24 add esp, 44
25 ret
}

int main(){
    int i=[10], *p.array0;
    p.array0=&a[0];
    initArray(a, 5);
    return *p.array0;
}

```

(b) Disassembly

Fig. 5. Interprocedural example

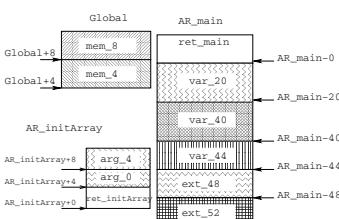


Fig. 6. Memory-regions

Formal Parameters. On entry to a procedure, `esp` points to the return address, and the parameters to the procedure are the bytes beyond the return address (in the positive direction). Hence the offsets for the formal parameters will be positive. Hence, a-locs with positive offsets are considered to be the formal parameters.

At a call on a procedure that has k formals, the last k extended a-locs represent the actual parameters.

Fig. 6 shows the extended a-locs for procedure `main` and the formal parameters for procedure `initArray` for the program in Example 3.

Handling of Calls and Returns. The interprocedural algorithm is similar to the intraprocedural algorithm, but analyzes the supergraph of the executable. In the supergraph, each call site has two nodes: a call node and an end-call node. The only successor of the call node is the entry node of the called procedure and the only predecessor of the end-call node is the exit node of the procedure called by the corresponding call node. The call→entry and the exit→end-call edges will be referred to as *linkage edges*. Nodes, edges and edge-transformers for all other instructions are similar to the intraprocedural CFG.

The transformer for the call→entry edge assigns actuals to formals and also changes `esp` to reflect the change in the current AR. First the join of the abstract stores at the call-sites of P is computed; then the value-set of `esp` in the newly computed value is set to $(\perp, \dots, 0, \dots, \perp)$, where the 0 occurs in the slot for P. In addition, each formal parameter `Formali` is initialized as follows:

$$(F_i^c, P_i^c) = *(\text{as}_c[\text{esp}] \boxplus (\text{offset}(\text{AR_P}, \text{Formal}_i) - 4), S_i)$$

$$\text{as}_{\text{enter_P}}[\text{Formal}_i] = \begin{cases} \top & \# \text{ if } \bigcup_{c \in \text{callsites}(P)} P_i^c \neq \phi \\ \bigsqcup_{c \in \text{callsites}(P), v \in F_i^c} \text{as}_c[v] & \# \text{ otherwise} \end{cases}$$

where as_c is the abstract store at call-site c of P and S_i is the size of Formal_i . That is, the value-set for a formal on entry is the join of the value-sets of the corresponding actuals at the callers. The offset of the actual in the AR of the caller is determined from the offset of the formal parameter. In the fixpoint solution for Example 3, the abstract store for the enter node of `initArray` is: $\{\text{mem_4} \mapsto (0, \perp, \perp), \text{mem_8} \mapsto (1, \perp, \perp), \text{arg_0} \mapsto (\perp, -40, \perp), \text{arg_4} \mapsto (5, \perp, \perp), \text{eax} \mapsto (\perp, -40, \perp), \text{esp} \mapsto (\perp, \perp, 0), \text{ext_48} \mapsto (5, \perp, \perp), \text{ext_52} \mapsto (\perp, -40, \perp)\}$. The regions in the value-sets are listed in the following order: (`Global`, `AR_main`, `AR_initArray`).

The transformer for the exit→end-call edge ordinarily restores the value-set of `esp` to the value before the call. This corresponds to the normal case when the callee restores the value of `esp` to the value before the call. However, in some procedures the callee does not restore `esp`. For instance, `alloca` allocates memory on the stack by subtracting some number of bytes from `esp`. VSA takes care of those changes in `esp` that are just additions/subtractions to the initial value when it can determine that the change is always some constant amount. In such cases, `esp` is restored to the value before the call plus/minus the change. If VSA cannot determine that the change is a constant, then it issues an error report.

5 Affine Relations

Recall that in Example 2, VSA was unable to find finite upper bounds for `eax` at instruction 7 and `ebx` at instruction 9. This causes `ret_main` to be added to the possibly-killed sets for instructions 7 and 9. This section describes how our implementation of

VSA obtains improved results, by identifying and then exploiting integer affine relations that hold among the program’s registers, using an interprocedural algorithm for affine-relation analysis due to Müller-Olm and Seidl [19]. The algorithm is used to determine, for each program point, all affine relations that hold among an x86’s 8 registers. More details about the algorithm can be found in [19].

An integer affine relation among variables r_i ($i = 1 \dots n$) is a relationship of the form $a_0 + \sum_{i=1}^n a_i r_i = 0$, where the a_i ($i = 1 \dots n$) are integer constants. An affine relation can also be represented as an $(n+1)$ -tuple, (a_0, a_1, \dots, a_n) . There are two opportunities for incorporating information about affine relations: (i) in the interpretation of conditional instructions, and (ii) in an improved widening operation. Our implementation of VSA incorporates both of these uses of affine relations.

At instruction 14 in the program in Fig. 1, `eax`, `esp`, and `ecx` are all related by the affine relation $\text{eax} = (\text{esp} + 4 \times \text{ecx}) + 4$. When the true branch of the conditional `j1` at L1 is interpreted, `ecx` is bounded on the upper end by 4, and thus the value-set `ecx` at L1 is $([0, 4], \perp)$. (A value-set in which all RICs are \perp except the one for the `Global` region represents a set of pure numbers, as well as a set of global addresses.) In addition, the value-set for `esp` at L1 is $(\perp, -44)$. Using these value-sets and solving for `eax` in the above relation yields

$$\text{eax} = (\perp, -44) + 4 \times ([0, 4], \perp) + 4 = (\perp, -44) + 4 \times [0, 4] + 4 = (\perp, 4[0, 4] - 40).$$

In this way, a sharper value for `eax` at L1 is obtained than would otherwise be possible; Such bounds cannot be obtained for loops that are controlled by a condition that is not based on indices; however, the analysis is still safe in such cases.

Halbwachs et al. [15] introduced the “widening-up-to” operator (also called *limited widening*), which attempts to prevent widening operations from “over-widening” an abstract store to $+\infty$ (or $-\infty$). To perform limited widening, it is necessary to associate a set of inequalities M with each widening location. For polyhedral analysis, they defined $P\nabla_M Q$ to be the standard widening operation $P\nabla Q$, together with all of the inequalities of M that satisfy both P and Q . They proposed that the set M be determined by the linear relations that force control to remain in the loop. Our implementation of VSA incorporates a limited-widening algorithm, adapted for reduced interval congruences. For instance, suppose that $P = (x \mapsto 3[0, 2] + 5)$, $Q = (x \mapsto 3[0, 3] + 5)$, and $M = \{x \leq 28\}$. Ordinary widening would produce $(x \mapsto 3[0, +\infty] + 5)$, whereas limited widening would produce $(x \mapsto 3[0, 7] + 5)$. In some cases, however, the a-loc for which VSA needs to perform limited widening is a register r_1 , but not the register that controls the execution of the loop (say r_2). In such cases, the implementation of limited widening uses the results of affine-relation analysis—together with known constraints on r_2 and other register values—to determine constraints that must hold on r_1 . For instance, if the loop back-edge has the label $r_2 \leq 20$, and affine-relation analysis has determined that $r_1 = 4 * r_2$ always holds at this point, then the constraint $r_1 \leq 80$ can be used for limited widening of r_1 ’s abstract store.

The performance evaluation in §7 uses a version of affine-relation analysis that models the restoration of callee-save registers across calls. (At present, certain technical difficulties preclude a similar treatment of caller-save registers. We have also not yet implemented a check to determine that the code obeys the calling conventions for caller-save and callee-save registers.)

Table 1. Running times and storage requirements for VSA and affine-relation analysis.

Program	Procedures	Instructions	Malloc sites	Indirect jumps	Calls	Indirect calls	Memory usage (MB)	Value-set analysis (sec.)	Affine-relation analysis (sec.)
javac	36	3555	1	0	133	79	51	76	29
cat (2.0.14)	123	3892	1	3	138	4	42	9	26
cut (2.0.14)	129	4329	2	3	182	4	48	7	42
grep (2.4.2)	245	16808	18	4	654	6	102	117	75
gcc (2.96)	252	22984	8	3	1048	4	232	108	295
tar (1.13.19)	581	47739	11	21	2553	29	258	220	156
awk (3.1.0)	595	69927	84	33	3669	152	623	1017	1011
winhlp32 (5.0.2195.2014)	1018	108380	0	10	6002	1005	737	1712	1290

6 Indirect Jumps and Indirect Calls

The supergraph of the program will not be complete in the presence of indirect jumps and indirect calls. Consequently, missing jump and call edges need to be inserted during VSA. For instance, suppose that VSA is interpreting an indirect jump instruction $J1: \text{jmp } 1000[\text{eax}*4]$, and let the current abstract store at this instruction be $\{\text{eax} \mapsto ([0, 9], \perp, \dots, \perp)\}$. Edges need to be added from $J1$ to the instructions whose addresses could be in memory locations $\{1000, 1004, \dots, 1036\}$. If the addresses $\{1000, 1004, \dots, 1036\}$ refer to the read-only section of the program, then the addresses of the successors of $J1$ can be read from the header of the executable. If not, the addresses of the successors of $J1$ in locations $\{1000, 1004, \dots, 1036\}$ are determined from the current abstract store at $J1$. Due to possible imprecision in VSA, it could be the case that VSA reports that the locations $\{1000, 1004, \dots, 1036\}$ have all possible addresses. In such cases, VSA proceeds without adding new edges. However, this could lead to an under-approximation of the value-sets at program points. Therefore, the analysis issues a report to the user whenever such decisions are made. We will refer to such instructions as *unsafe instructions*. Another issue with using the results of VSA is that an address identified as a successor of $J1$ might not be the start of an instruction. Such addresses are ignored, and the situation is reported to the user.

Indirect calls are handled similarly, with a few additional complications.

- A successor instruction identified by the method outlined above may be in the middle of a procedure. In such cases, the analysis reports this to the user.
- The successor instruction may not be part of a procedure that was identified by IDAPro. This is due to the limitations of IDAPro’s procedure-finding algorithm: IDAPro does not identify procedures that are called exclusively via indirect calls. In such cases, VSA can invoke IDAPro’s procedure-finding algorithm explicitly, to force a sequence of bytes from the executable to be decoded into a sequence of instructions and spliced into the IR for the program. (At present, this technique has not yet been incorporated in our implementation.)

7 Performance Evaluation

Table 1 shows the running times and storage requirements of our prototype implementation for analyzing a set of Win32 and Linux/x86 programs; the program version is

Table 2. Comparison of 3 variants of VSA.

Program	No VSA	VSA	Crude VSA
javac	21597	52884	54996
cat (2.0.14)	17932	32826	33632
cut (2.0.14)	23116	37834	39116
grep (2.4.2)	123293	201584	217003
gcc (2.96)	320089	5921020	5970559
tar (1.13.19)	644518	4088659	4305446

shown in parentheses. As a temporary expedient, calls to library functions are treated during analysis as identity transformers. The analyses were performed on a Pentium-4 with a clock speed of 3.06GHz, equipped with a physical memory of 4GB and running Windows 2000. (The per-process address space was limited to 2GB.)

To contrast the capabilities of VSA with analysis algorithms that treat memory accesses very conservatively—i.e., if a register is assigned a value from memory, it is assumed to take on any value—we compared it with a version of VSA, called *crude VSA*, that always sets the value-sets for all non-register a-locs to \top . Table 2 shows the number of flow-dependence edges obtained with three methods: (i) without using VSA at all (which causes dependences to be missed); (ii) with VSA; and (iii) with crude VSA.

8 Soundness Issues

Soundness would mean that value-set analysis would identify used, killed, and possibly-killed sets that would never miss any data dependence, although they might cause spurious dependences to be reported. This is a lofty goal; however, it is not clear that a tool that achieves this goal would have practical value. There are less lofty goals that do not meet this standard—but may result in a more practical system. In particular, we may not care if the system is sound, as long as it can provide warnings about the situations that arise during the analysis that threaten the soundness of the results. This is the path that we are following in our work.

Here are some of the cases in which the analysis can be unsound, but where the system generates a report about the nature of the unsoundness:

- The program is vulnerable to a buffer-overrun attack. This can be detected by identifying a point at which there can be a write past the end of a memory-region.
- The control-flow graph and call-graph may not identify all successors of indirect jumps and indirect calls. Report generation for such cases is discussed in §6.
- A related situation is a jump to a code sequence concealed in the regular instruction stream; the alternative code sequence would decode as a legal code sequence when read out-of-registration with the instructions in which it is concealed. The analysis could detect this situation as an anomalous jump to an address that is in the code segment, but is not the start of an instruction.
- With self-modifying code, the control-flow graph and call-graph are not available for analysis. The analysis can detect the possibility that the program is self-modifying by identifying an anomalous jump or call to a modifiable location.

9 Related Work

There is an extensive body of work on analyzing executables. The work that is most closely related to VSA is the alias-analysis algorithm for executables proposed by Debray et al. [11]. The basic goal of their algorithm is similar to that of VSA: for them, it is to find an over-approximation of the set of values that each register can hold at each program point; for us, it is to find an over-approximation of the set of values that each (abstract) data object can hold at each program point, where data objects include memory locations in addition to registers. In their analysis, a set of addresses is approximated by a set of congruence values: they keep track of only the low-order bits of addresses. However, unlike our algorithm, their algorithm does not make any effort to track values that are not in registers. Consequently, they lose a great deal of precision whenever there is a load from memory.

Cifuentes and Fraboulet [5] give an algorithm to identify an intraprocedural slice of an executable by following the program’s use-def chains. However, their algorithm also makes no attempt to track values that are not in registers, and hence cuts short the slice when a load from memory is encountered.

Past work on decompiling assembly code to a high-level language is also related to our goals [6,4,20]. However, that work has also not done much to address the problem of recovering information about memory accesses.

The idea of inferring the layout of a program’s data structures based on the access patterns in the program is similar to the idea behind the Aggregate Structure Identification (ASI) algorithm of Ramalingam et al. [24]. However, ASI cannot be applied to x86 code without having the results of VSA already in hand: ASI requires points-to, range, and stride information; however, this information is not available for an x86 executable until after VSA. The good news is that ASI can be applied after VSA to refine the program’s a-locs, which can allow some clients of value-set analysis—such as dependence analysis—to compute more precise results. We plan to use ASI in conjunction with the results of value-set analysis in future work.

Xu et al. [31] also created a system that analyzed executables in the absence of symbol-table and/or debugging information. The goal of their system was to establish whether or not certain memory-safety properties held in SPARC executables. Initial inputs to the untrusted program were annotated with typestate information and linear constraints. The analyses developed by Xu et al. were based on classical theorem-proving techniques: the typestate-checking algorithm used the induction-iteration method [30] to synthesize loop invariants and Omega [23] to decide Presburger formulas. In contrast, the goal of the system described in the present paper is to recover information from an x86 executable that permits the creation of intermediate representations similar to those that can be created for a program written in a high-level language. VSA uses abstract-interpretation techniques to determine used, killed, and possibly-killed sets for each instruction in the program.

Several people have developed techniques to analyze executables in the presence of additional information, such as the source code, symbol-table information, or debugging information [18,2,1,27]. Analysis techniques that assume access to such information are limited by the fact that it must not be relied on when dealing with programs such as viruses, worms, and mobile code (even if such information is present).

Dor et al. [12] present a static-analysis technique—implemented for programs written in C—whose aim is to identify string-manipulation errors, such as potential buffer

overruns. In their work, a flow-insensitive pointer analysis is first used to detect pointers to the same base address; integer analysis is then used to detect relative-offset relationships between values of pointer variables. The original program is translated to an integer program that tracks the string and integer manipulations of the original program; the integer program is then analyzed to determine relationships among the integer variables, which reflect the relative-offset relationships among the values of pointer variables in the original program. Because they are primarily interested in establishing that a pointer is merely *within the bounds* of a buffer, it is sufficient for them to use linear-relation analysis [10], in which abstract stores are convex polyhedra defined by linear inequalities of the form $\sum_{i=1}^n a_i x_i \leq b$, where b and the a_i are integers, and the x_i are integer variables.

In our work, we are interested in discovering fine-grained information about the structure of memory-regions. As already discussed in §3.3, it is important for the analysis to discover alignment and stride information so that it can interpret indirect-addressing operations that implement field-access operations in an array of structs or pointer-dereferencing operations. Because we need to represent non-convex sets of numbers, linear-relation analysis is not appropriate. For this reason, the numeric component of VSA is based on reduced interval congruences, which are capable of representing certain non-convex sets of integers.

Rugina and Rinard [28] have also used a combination of pointer and numeric analysis to determine information about a program’s memory accesses. There are several reasons why their algorithm is not suitable for the problem that we face: (i) Their analysis assumes that the program’s local and global variables are known before analysis begins: the set of “allocation blocks” for which information is acquired consists of the program’s local and global variables, plus the dynamic-allocation sites. (ii) Their analysis determines range information, but does not determine alignment and stride information. (iii) Pointer and numeric analysis are performed separately: pointer analysis is performed first, followed by numeric analysis; moreover, it is not obvious that pointer analysis could be intertwined with the numeric analysis that is used in [28].

Our analysis *combines* pointer analysis with numeric analysis, whereas the analyses of Rugina and Rinard and Dor et al. use two separate phases: pointer analysis *followed by* numeric analysis. An advantage of combining the two analyses is that information about numeric values can lead to improved tracking of pointers, and pointer information can lead to improved tracking of numeric values. In our context, this kind of positive interaction is important for discovering alignment and stride information (cf. §3.3). Moreover, additional benefits can accrue to clients of VSA; for instance, it can happen that extra precision will allow VSA to identify that a strong update, rather than a weak update, is possible (i.e., an update can be treated as a kill rather than as a possible kill; cf. case two of Fig. 4). The advantages of combining pointer analysis with numeric analysis have been studied in [22]. In the context of [22], combining the two analysis only improves precision. However, in our context, a combined analysis is needed to ensure safety.

References

1. J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng.*, 2001.

2. J. Bergeron, M. Debbabi, M.M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE*, pages 184–189, 1999.
3. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, Lec. Notes in Comp. Sci. Springer-Verlag, 1993.
4. C. Cifuentes and A. Fraboulet. Interprocedural data flow recovery of high-level language code from assembly. Technical Report 421, Univ. Queensland, 1997.
5. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Int. Conf. on Softw. Maint.*, pages 188–195, 1997.
6. C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Int. Conf. on Softw. Maint.*, pages 228–237, 1998.
7. CodeSurfer, GrammaTech, Inc., <http://www.grammatach.com/products/codesurfer/>.
8. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Int. Symp. on Programming*, pages 106–130. Dunod, Paris, France, 1976.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Princ. of Prog. Lang.*, 1977.
10. P. Cousot and R. Cousot. Automatic discovery of linear restraints among variables of a program. In *Princ. of Prog. Lang.*, pages 84–97, 1978.
11. S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Princ. of Prog. Lang.*, pages 12–24, 1998.
12. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Prog. Lang. Design and Impl.*, pages 155–167, 2003.
13. Fast library identification and recognition technology, DataRescue sa/nv, Liège, Belgium, <http://www.datarescue.com/idabase/flirt.htm>.
14. P. Granger. Static analysis of arithmetic congruences. *Int. J. of Comp. Math.*, 1989.
15. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
16. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.
17. IDAPro disassembler, <http://www.datarescue.com/idabase/>.
18. J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Prog. Lang. Design and Impl.*, pages 291–300, 1995.
19. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Princ. of Prog. Lang.*, 2004.
20. A. Mycroft. Type-based decompilation. In *European Symp. on Programming*, 1999.
21. E.W. Myers. Efficient applicative data types. In *Princ. of Prog. Lang.*, pages 66–75, 1984.
22. A. Pioli and M. Hind. Combining interprocedural pointer analysis and conditional constant propagation. Tech. Rep. RC 21532(96749), IBM T.J. Watson Research Center, March 1999.
23. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.
24. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Princ. of Prog. Lang.*, pages 119–132, 1999.
25. T. Reps and G. Rosay. Precise interprocedural chopping. In *Found. of Softw. Eng.*, 1995.
26. T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *Trans. on Prog. Lang. and Syst.*, 5(3):449–477, July 1983.
27. X. Rival. Abstract interpretation based certification of assembly code. In *Int. Conf. on Verif., Model Checking, and Abs. Int.*, 2003.
28. R. Rugini and M.C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. New York, NY. ACM Press.
29. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
30. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Princ. of Prog. Lang.*, pages 132–143, 1977.
31. Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. In *Prog. Lang. Design and Impl.*, pages 70–82, 2000.

The Limits of Alias Analysis for Scalar Optimizations

Rezaul A. Chowdhury¹, Peter Djeu¹, Brendon Cahoon²,
James H. Burrill³, and Kathryn S. McKinley¹

¹ Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA,
`{shaikat, djeu, mckinley}@cs.utexas.edu`
² Conformative Systems, Austin, TX 78759, USA,
`brendon.cahoon@conformative.com`

³ Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA,
`burrill@cs.umass.edu`

Abstract. In theory, increasing alias analysis precision should improve compiler optimizations on C programs. This paper compares alias analysis algorithms on scalar optimizations, including an analysis that assumes no aliases, to establish a very loose upper bound on optimization opportunities. We then measure optimization opportunities on thirty-six C programs. In practice, the optimizations are rarely inhibited due to the precision of the alias analyses. Previous work finds similarly that the increased precision of specific alias algorithms provide little benefit for scalar optimizations, and that simple static alias algorithms find almost all dynamically determined aliases. This paper, however, is the first to provide a static methodology that indicates that additional precision is unlikely to yield improvements for a set of optimizations. For clients with higher alias accuracy demands, this methodology can help pinpoint the need for additional accuracy.

1 Introduction

An enormous amount of research develops compiler alias analysis for C programs, which determines if two distinct variables may reference the same memory location [1,5,9,16, 23,28,29,34,35]. The literature shows an essential tradeoff: improving the precision of alias analysis increases the cost of performing it, and this increase can be substantial. In theory, a more precise alias analysis improves the clients' results. The clients of alias analysis are numerous, and include improving program performance [17,34], finding bugs [18], and pinpointing memory leaks [19]. In some cases the theory holds true. For instance, automatic parallelization [34] and error detection [18] benefit from the judicious application of precise alias analysis.

This paper studies alias analysis on scalar compiler optimizations designed to improve performance. We implement three analyses: address-taken, Steensgaard [29], and Shapiro-Horwitz [27]. We also introduce an upper bound methodology that assumes there are no alias relations, and thus the compiler is never inhibited by an alias when applying optimizations. Previous alias analysis evaluation typically counts the static number of aliases and assumes fewer is better. This upper bound is not guaranteed to be tight, and is useful only for static evaluation.

We use nine scalar optimizations to compare the alias analyses to this static upper bound on thirty-six C programs from popular benchmark suites. The optimizations are

sparse conditional constant propagation, global variable replacement, loop unrolling, loop invariant code motion, global value numbering, copy propagation, useless copy removal, array strength reduction, and scalar replacement. We measure optimization applications individually and as a group. Experiments show there is only a very small gap between Shapiro-Horwitz and the static upper bound. The increased precision of Shapiro-Horwitz attains minor improvements over Steensgaard with respect to optimization, and both are somewhat better than address-taken. The largest difference is for loop invariant code motion, for which the upper bound methodology detects on average six percent more loop invariant computations than the best analysis.

A few other studies on the effect of alias analysis on scalar optimizations also suggest that a simple alias analysis will suffice [3, 12, 14, 15, 21]. For example, Hind and Pioli show that few additional scalar analysis opportunities come from increasing alias precision on twenty-three C programs [21]. Diwan et al. measure dynamically the additional opportunities for two optimizations on Modula-3 programs and find that improving alias analysis will not benefit these optimizations much, if at all [14]. Das et al. measure aliases dynamically, without respect to an optimization, and find that a simple analysis misses 5% of independent memory accesses in nine C programs [12].

Das et al. suggest the following [12]: “Ideally, we would like to repeat their study [Diwan et al.] for every conceivable optimization and every pointer analysis.” Although we of course do not study “every conceivable optimization,” the range here is more numerous than previous work. The most important contribution of this paper, however, is obviating the need for evaluating all pointer analyses. For thirty-six C programs, no matter how much additional precision an alias analysis provides over Steensgaard or Shapiro-Horwitz, that extra precision is unlikely to benefit scalar optimizations. Other clients with higher precision needs can also use this methodology to pinpoint opportunities for improvement.

The remainder of this paper is organized as follows. Section 2 compares our methodology to previous pointer analysis research. Section 3 overviews the alias analyses: address-taken, Steensgaard [29], Shapiro-Horwitz [27], and the no aliases upper bound. Section 4 and 5 introduces the *Scale* compiler framework and optimizations. Section 6 presents the experimental methodology. Section 7 shows measurements of optimization opportunities and compile times, demonstrating that additional alias precision will not yield many, if any, improvements to scalar optimizations and thus performance.

2 Related Work

This section describes comparative studies of alias analyses. We focus on the closest related work that use the clients of the alias analyses for evaluation. The evaluation of most new alias or points-to analysis algorithms reports the size of the static points-to information they compute. For example, Emami et al. [16], and Wilson and Lam [34] introduce new context-sensitive interprocedural points-to algorithms and evaluate them using the size of the points-to sets.

Other researchers evaluate alias analysis algorithms by reporting changes to the size of the static points-to information when the precision of the algorithm changes. Ruf evaluates the effect of context-sensitivity on the precision of alias analysis [25]. Ruf

concludes that adding context-sensitivity does not improve the precision for the benchmarks he examines. Liang and Harrold introduce a context-sensitive flow-insensitive algorithm, and they compare their algorithm to three other algorithms [23]. Yong et al. present a tunable pointer analysis framework that distinguishes fields structures [35].

Hind and Pioli focus on the client and compare five alias analysis algorithms using scalar analyses and optimizations [21]. They use Mod/Ref analysis, live variable analysis, reaching definitions analysis, and interprocedural constant propagation. We corroborate their results, but we do so within the context of a new compiler and with a focus on a more comprehensive selection of client optimizations, rather than analyses. In earlier work, Hind and Pioli present an empirical comparison of four alias analysis algorithms with different levels of flow sensitivity [20]. They measure the precision of the analysis results, and the time and space to compute the results. They do not study the effect of analysis quality on optimizations in this work.

Shapiro and Horwitz compare the precision of four flow and context-insensitive pointer analysis algorithms [27,28]. They test the precision of the pointer analyses using GMOD analysis, live variable analysis, truly live variable analysis, and an interprocedural slicing algorithm. Shapiro and Horwitz conclude that more precise analysis does improve the results of some, but not all of the clients. Stocks et al. compare the flow-sensitive and context-sensitive analysis on Mod analysis [30]. They conclude that more precision helps improve the precision of Mod analysis. These two papers are focused on analysis clients rather than the optimization clients we use.

Diwan et al. evaluate three alias analysis algorithms using static, dynamic, and upper bound metrics [13,14,15]. They demonstrate the effect of the three analyses using redundant load elimination and method invocation resolution. They show that a fast and simple alias analysis is effective for type-safe languages. Bacon and Sweeney find similar results for C++ method resolution [3].

Cooper and Lu use pointer analysis to perform register promotion, which is an optimization that converts references to scalar values in memory to a register [10]. Identifying aliases is important for this optimization, but Cooper and Lu do not show how the precision of the analysis affects optimization opportunities.

Ghiya and Hendren empirically show that their points-to analysis and connection analysis can improve loop-invariant removal and common subexpression elimination, array dependence testing, and program understanding [17]. They do not experiment with the precision of the analysis, and they concede that a conservative analysis may provide the same benefits for the scalar optimizations.

Das et al. measure the effect of pointer analysis on optimizations [12]. Their goal is to evaluate whether flow-insensitive pointer analysis is sufficient for compiler optimizations. Das et al. do not use any specific optimization or compiler, but instead develop a new metric for evaluating the precision of pointer analysis.

Our work is in the spirit of the last four studies, all of which focus on the client optimizations. We are, however, broader in scope in terms of the range of optimizations and the number of programs. In addition, we use a new methodology that computes a static upper bound that shows, for our programs and optimizations, that no additional precision is needed.

3 Alias Analysis

We study the following alias analysis algorithms (1) Address-taken, (2) Steensgaard [29], (3) Shapiro-Horwitz [28], and (4) Assume no aliases. Address-taken is very simple and is linear in the size of the input program. The compiler assumes all heap objects are potential aliases of each other, and includes in this set all variables for which the program explicitly takes their address. The address-taken algorithm produces the most conservative set of alias relations.

Steengaard's algorithm is interprocedural and flow-insensitive. It has almost linear running time and linear space complexity, but does not necessarily produce precise results [29]. It is based upon type-inference methods using alias relations. It results in alias sets that are symmetric and transitive. The Shapiro-Horwitz algorithm [28] extends and increases the precision of Steengaard's algorithm without a significant effect on running time. A parameter specifies the precision between the lowest precision (Steengaard) to the highest precision (Andersen's algorithm [1]). The analysis time varies inversely with precision. We choose an intermediate point for our evaluation.

Assuming no aliases serves as a static metric for evaluating the effect of alias information on clients. It simply communicates the empty set of alias relations to the optimizations. Since the compiler makes the sometimes false assumption that there are no aliases, the generated executable can be incorrect. The purpose of this analysis is thus not to generate a working executable, but to establish a loose upper bound for the maximum number of optimizations the compiler could perform.

4 The Scale Compilation System

This section outlines the Scale compilation framework, and its representation of aliases. The subsequent section enumerates the client optimizations and how they use aliases.

Scale is a flexible, high performance research compiler for C and Fortran, and is written in Java [24,32]. Scale transforms programs into a control flow graph, performs alias analysis, and uses the results to build a static single assignment (SSA) [11], machine-independent intermediate representation (IR) that we call *Scribble*. Scale performs optimizations on Scribble, and then transforms Scribble to a low-level, more machine dependent RISC style IR on which it performs a variant of linear scan register allocation [31]. It outputs C or assembly for the Alpha and Sparc processors.

Scale transforms the control flow graph (CFG) to SSA form after it performs alias analysis. SSA form ensures that each use of a scalar variable, or a virtual variable created during pointer analysis, gets its value from a single definition [11]. Scale utilizes Chow et al.'s technique for representing pointers, which makes a distinction between definitions that must occur and may occur [7]. Chow et al. define virtual variables to represent indirect variables (e.g., `*p`). Chow et al. create a unique virtual variable for all indirect variables that have similar alias characteristics. They perform alias analysis on the virtual variables and the scalar variables. Scale's analysis has a subtle difference; it performs alias analysis prior to creating the virtual variables. After alias analysis, Scale defines a unique virtual variable for each alias group, which are sets of variables that share the same aliases.

In Scale, the SSA form thus includes may and must definitions that are linked to uses by corresponding edges. Optimizations traverse these edges to find definitions, recurrences, etc. All scalar optimizations (except useless copy removal) manipulate the SSA form of the control flow graph. The precision of disambiguation information derived from alias analysis thus directly impacts the quality of the SSA graph, and consequently optimization opportunities and results.

5 Scalar Optimizations

This section describes each optimization and the optimization success *criteria* that we measure and report. Scale performs scalar optimizations on SSA Scribble form except for useless copy removal. The optimizations target scalar variables, loads, scalar expressions, array address arithmetic, and heap allocated arrays. We expect that alias analysis will have more effect on additional optimizations that specifically target heap pointers.

Loop Invariant Code Motion (LICM). LICM finds computations (including loads) in loops that produce the same value on every iteration and moves them to appropriate locations outside the loop. For nested loops, it moves computation out of as many inner-loops as possible without destroying program semantics. LICM thus reduces the number of instructions executed. In Scale, SSA use-def links indicate where the CFG node gets its definitions, and LICM moves computations to the outer-most basic block in which the definition is available. More precise alias information can reveal additional invariant expressions. To preserve program semantics, Scale only moves stores for local variables. It never moves procedure calls or expressions involving global variables.

Criteria: number of expressions moved.

Sparse Conditional Constant Propagation (SCCP). SCCP discovers variables and expressions that are constant and propagates them throughout the program. SCCP correctly propagates constants even in the presence of conditional control flow. It speeds up program execution by evaluating expressions at compile time instead of run time and improves the effectiveness of other optimizations, such as value numbering. Scale uses Wegman and Zadeck’s SCCP algorithm on SSA-form [33]. Scale uses alias analysis to obtain variable values in the presence of pointer operations. More precise alias analysis can thus reveal more constants.

Criteria: number of constants propagated.

Copy Propagation (CP). CP discovers assignments of the form $x \leftarrow y$ and replaces any later use of variable x by y when no intervening instruction changes x or y . CP then removes the original assignment statement. Scale does not propagate a copy if (1) the right-hand-side variable of the assignment statement contains May-Use information indicating that it may be involved in an alias relationship, or (2) if either argument in the assignment is a global variable.

Criteria: number of copies propagated.

Global Value Numbering (GVN). GVN determines whether two computations are equivalent and if so, removes one of them. Scale uses the dominator tree-based value numbering technique by Briggs et al. [4]. It assigns a *value number* to each computation

and exposes equivalences when it assigns distinct computations the same value number. SSA form simplifies this process. GVN works on entire procedures instead of single basic blocks, as in traditional value numbering. It improves program running time by removing redundant computations.

Criteria: number of expressions removed.

Loop Unrolling (LU). LU replaces the body of a loop by several copies of the body and adjusts the loop control code accordingly. Aliases inhibit loop unrolling only if the loop control variables may be aliased with loop varying variables. LU reduces the number of instructions executed during run time at the cost of increased code size, and may improve the effectiveness of other optimizations, such as common-subexpression elimination and strength reduction.

Criteria: number of loops unrolled.

Scalar Replacement (SR). Register allocators usually do not allocate subscripted variables to registers. Scalar Replacement tricks the allocator by replacing subscripted variables with scalars and thus making them available for register allocation. Dependence analysis locates reuse of array elements and then SR replaces them with assignments and uses of scalar temporaries. SR reduces the number of loads and stores in programs and is very effective in reducing execution times.

Criteria: number of array loads replaced.

Global Variable Replacement (GVR). GVR replaces references to global variables with references to local variables by copying the global into a local only when the global is not aliased to another variable that the procedure modifies.

Criteria: number of loads to global variables replaced.

Array Access Strength Reduction (AASR). AASR uses the *method of finite differences* to replace expensive operators in array element address calculations with cheaper ones. Scale targets array index calculations in the inner-most loops, and replaces multiplications with additions when possible. It moves any resulting loop invariants outside the loop and folds constant expressions as part of this process.

Criteria: number of array index calculations replaced.

Useless Copy Removal - (UCR). UCR removes copy statements of the form $x \leftarrow x$ in the CFG form. Scale creates these statements when transitioning to and from SSA and via other optimizations. Because transitioning out of SSA form introduces copies and new temporary variables based on the SSA edges, UCR is sensitive to edges induced by alias analysis and other optimizations.

Criteria: number of useless copies removed.

6 Methodology

Table 1 enumerates our test suite programs from the following benchmark suites: SPEC 95, SPEC 2000, Austin from Todd Austin [2], McCAT from McGill [16], and Landi-PROLANGS from Rutgers [22,26,25]. Our test suite closely follows Hind and Pilio's [21], and all but two of their programs appear in our study. We omit 052.alvinn

Table 1. Benchmark suites

Abbr.	Benchmark suite
A	Austin's
MC	McCAT
LP	Landi-PROLANGS
S95	SPEC 95
SO0	SPEC 2000

Table 2. Alias analysis algorithms

Abbr.	Algorithm
AT	Address-taken analysis
ST	Steensgaard's interprocedural algorithm
SH-4	Shapiro-Horwitz's interprocedural algorithm with 4 categories
NA	Assume no aliases

Table 3. Optimizations

Abbr.	Scale option	Optimization
AASR	a	Array Access Strength Reduction
SCCP	c	Sparse Conditional Constant Propagation
GVR	g	Global Variable Replacement
LU	j	Loop Unrolling
LICM	m	Loop Invariant Code Motion
GVN	n	Global Value Numbering
CP	p	Copy Propagation
UCR	u	Useless Copy Removal
SR	x	Scalar Replacement

from SPEC 92 because the SPEC 2000 versions subsume them. We omit 17.binr from McCAT because of a Scale compilation bug.

We use the October 2003 development version of Scale with the default parameters except as noted. We specify the alias analysis from Table 2. (Shapiro-Horwitz with one category behaves the same as Steensgaard although the implementations are distinct). We select four categories as the input parameter to Shapiro-Horwitz so that it behaves as an intermediate point that is more precise than Steensgaard, but not as expensive as Andersen [1]. We either select a fixed sequence of optimizations or choose a single optimization and turn off the others. Table 3 enumerates the optimizations, their Scale option letter, and our abbreviation.

Scale also implements Partial Redundancy Elimination (PRE) using Chow et al.'s algorithm for SSA [6]. This algorithm requires SSA form, but does not produce SSA, which makes it difficult to measure and use in Scale. Furthermore, our PRE results show more optimization opportunities with Steensgaard and Shapiro-Horwitz than with no aliases. We believe this anomaly results from either a bug in Scale or an interaction with SSA. We omit the PRE results here since we believe the underlying problem is orthogonal to alias analysis. A companion technical report contains these results [8].

We measure compile times on a 502 MHz UltraSPARC-IIe Sun Blade 100 running SunOS 5.8 with 256 MB of RAM. Since our compiler is written in Java, we specify an initial heap size of 100 MB and a maximum heap size of 1000 MB for Sun's Java virtual machine running Scale.

7 Results

We vary the alias analysis and compare compilation times for each benchmark. For scalar optimization opportunities, the results summarize across benchmarks; a companion technical report [8] contains complete per program results.

7.1 Compile Time

Table 7.1 describes some characteristics of the 36 benchmark programs. The column marked “Src” identifies the benchmark suite to which the program belongs. The column marked “NCLC” reports the number of non-blank and non-commented lines of code in the program. The column marked “CFG Nodes” shows the number of nodes in the control flow graph created by Scale for the program. This number more accurately represents the program size as experienced by the compiler. The table arranges the programs in ascending order of the number of CFG nodes. The next three columns list the compile times (in seconds) of the program with all optimizations turned on in order “jgcamm-pxnmpu”. The first column uses address-taken analysis, the second uses Steensgaard’s interprocedural algorithm, and the third uses Shapiro-Horwitz’s interprocedural analysis with 4 categories. Each compile time is the smallest among 5 independent compiles of the same program with the same parameter values. Figure 1 shows the compile times from Table 7.1 as a bar graph.

The last two rows of Table 7.1 report the normalized average compile times over all the programs. We divide the compile time of the program for each alias analysis algorithm by the compile time of the program using address-taken analysis. Then, we take the arithmetic and geometric means (AM & GM) of those normalized compile times. The geometric mean reduces the effect of extreme values. The means suggest that using Steensgaard instead of address-taken increases the compile time by 5-6% on the average while the average increase in compile time due to the use of Shapiro-Horwitz is 20-30%. For large programs (like 186.crafty, 300.twolf, and 099.go), these percentages may grow to 150-200% and 350-400%, respectively. We believe this result is due to paging.

7.2 Optimization Opportunities

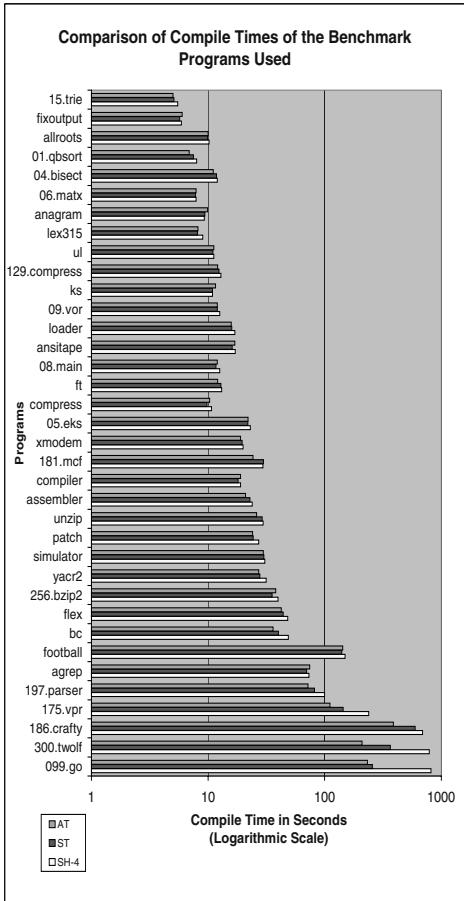
We measure the optimization opportunities utilized by each of the 9 scalar optimizations over each of the 36 benchmark programs over each of the 4 alias analyses based on the selected *criteria* (see Section 5). For each optimization, we report this criteria counter. Higher criteria counter values indicate more effective optimization results.

For a given optimization, we report the criteria counter using each of the 4 alias analyses and normalize the counters by dividing by the value obtained for NA (no aliases). To avoid dividing by zero if $numerator = denominator = 0$, we assume the normalized value is 1. If $numerator = 0 \neq denominator$, we set the *numerator* to 0.5 and proceed with the division. Although the case “ $numerator = denominator = 0$ ” occurs many times in our experiments, the case “ $numerator = 0 \neq denominator$ ” occurs only once: in 01.qsort for LICM in Table 6. We take the geometric mean of the normalized counters for all programs for each alias analysis.

We perform two sets of experiments. In the first set, we enable all 9 optimizations in the order “jgcamm-pxnmpu” (see Table 3) during each compilation. In the second set, we enable only one optimization per compilation. Table 5 summarizes the results for all optimizations enabled, and Table 6 for each one individually. Table 5 contains one row for each of the 9 optimizations and one column for each of the 4 alias analyses. In each optimization row, the each alias analysis column reports the geometric mean of

Table 4. Lines of code, CFG nodes, and compile times (in seconds).

Program	Src	NCLC	CFG nodes	Compile time (sec)		
				AT	ST	SH-4
15.trie	MC	311	197	5.0	5.1	5.5
fixoutput	LP	368	206	6.0	5.7	5.9
allroots	LP	155	272	10.0	9.9	10.2
01.qbsort	MC	200	294	6.9	7.5	8.0
04.bisect	MC	217	331	11.1	11.8	12.0
06.matx	MC	191	439	7.9	7.8	7.9
anagram	A	352	532	9.9	9.4	9.3
lex315	LP	598	658	8.2	8.1	9.0
ul	LP	472	773	11.2	11.0	11.2
129.compress	S95	1457	923	12.1	12.4	12.9
ks	A	585	987	11.6	10.9	10.9
09.vor	MC	984	1031	12.0	12.0	12.6
loader	LP	802	1082	15.8	16.0	17.0
ansitape	LP	1203	1113	17.0	16.1	17.1
08.main	MC	990	1115	12.0	11.7	12.6
ft	A	1113	1116	12.1	12.9	13.1
compress	LP	1071	1119	10.3	9.8	10.7
05.eks	MC	575	1498	22.0	21.9	23.0
xmodem	LP	1392	1718	19.0	19.6	20.0
181.mcf	S00	1482	1722	24.3	29.9	29.5
compiler	LP	2073	1789	19.0	18.1	19.0
assembler	LP	1891	2052	21.0	22.9	23.9
unzip	LP	2808	2637	26.1	29.1	29.7
patch	LP	2461	3248	24.1	24.4	27.2
simulator	LP	2881	3532	29.8	30.1	30.7
yacr2	A	2710	3753	27.2	27.8	31.5
256.bzip2	S00	3236	4888	38.0	35.4	39.8
flex	LP	4841	5405	42.5	44.1	48.2
bc	A	5449	5618	36.1	40.1	48.8
football	LP	1975	5765	143.1	140.4	149.7
agrep	LP	3434	8185	74.5	70.0	73.2
197.parser	S00	7921	15418	72.0	81.9	99.8
175.vpr	S00	11301	17935	111.0	144.2	238.8
186.crafty	S00	12985	22379	388.1	595.9	691.4
300.twolf	S00	17934	31414	209.3	365.9	788.1
099.go	S95	25895	35018	232.2	256.4	814.6
AM of norm. comp. times (wrt AT)		1.000	1.062	1.271		
GM of norm. comp. times (wrt AT)		1.000	1.052	1.190		

**Fig. 1.** Compile times.

the normalized optimization counters of all 36 programs. Each mean is subscripted by the average deviation of the counter values from that mean. For a given optimization, the column marked “Total for NA” contains the summation of the NA counter values for that optimization over all programs. The bar graph in Figure 2 graphs this table. To explain these results, we first define a sequence anomaly.

Sequence Anomaly: When the compiler applies a sequence of optimizations, the effectiveness of the optimizations later in the sequence is influenced by the type and number of opportunities exposed by earlier optimizations. Since optimizations interact with each other in a non-linear fashion, it is possible for a more precise alias analysis to have a negative impact on optimizations that come later in the sequence. We refer to these effects as *sequence anomalies*. Sequence anomalies cannot occur when only one optimization is applied to a program.

Table 5. Effectiveness of alias analysis on optimizations with all of them enabled in order “jgcamp-npxnmpu” (Geometric mean of normalized (w.r.t. NA) criteria counts with avg. deviation from the mean as subscript)

Opt.	Criteria	AT	ST	SH-4	NA	Total for NA
LU	loops unrolled	0.999 (0.002)	0.999 (0.002)	0.999 (0.002)	1.000 (0.000)	1004
GVR	loads replaced	0.999 (0.001)	0.999 (0.001)	0.999 (0.001)	1.000 (0.000)	16825
SCCP	constants propagated	0.967 (0.046)	0.970 (0.043)	0.970 (0.043)	1.000 (0.000)	25258
AASR	calculations replaced	0.992 (0.006)	0.999 (0.003)	0.999 (0.003)	1.000 (0.000)	7996
LICM	expressions moved	0.912 (0.123)	0.940 (0.086)	0.940 (0.086)	1.000 (0.000)	2136
GVN	expressions removed	0.979 (0.030)	0.992 (0.010)	0.992 (0.010)	1.000 (0.000)	24054
CP	copies propagated	0.978 (0.032)	0.987 (0.020)	0.987 (0.020)	1.000 (0.000)	21247
SR	array loads replaced	0.999 (0.002)	0.999 (0.001)	0.998 (0.003)	1.000 (0.000)	8143
UCR	useless copies removed	1.017 (0.025)	1.007 (0.012)	1.006 (0.012)	1.000 (0.000)	101543

Table 6. Effectiveness of alias analysis on optimizations with only one optimization enabled at a time (Geometric mean of normalized (w.r.t. NA) criteria counts with avg. deviation from the mean as subscript)

Opt.	Criteria	AT	ST	SH-4	NA	Total for NA
LU	loops unrolled	0.999 (0.002)	0.999 (0.002)	0.999 (0.002)	1.000 (0.000)	1004
GVR	loads replaced	1.000 (0.000)	1.000 (0.000)	1.000 (0.000)	1.000 (0.000)	10701
SCCP	constants propagated	0.948 (0.069)	0.948 (0.070)	0.948 (0.070)	1.000 (0.000)	9357
AASR	calculations replaced	0.993 (0.013)	0.993 (0.013)	0.993 (0.013)	1.000 (0.000)	2675
LICM	expressions moved	0.924 (0.101)	0.924 (0.101)	0.924 (0.101)	1.000 (0.000)	1236
GVN	expressions removed	0.953 (0.061)	0.985 (0.020)	0.985 (0.020)	1.000 (0.000)	13273
CP	copies propagated	0.968 (0.041)	0.968 (0.041)	0.968 (0.041)	1.000 (0.000)	9203
SR	array loads replaced	0.998 (0.003)	0.999 (0.002)	1.000 (0.001)	1.000 (0.000)	5135

7.3 Optimizations in Sequence

We first summarize the average effect of AT, ST, and SH-4 using a fixed sequence of optimizations, and then summarize the results for each particular optimization.

Address-taken: For LU, GVR, and SR, the effectiveness of address-taken analysis is within 0.1% of that of any alias analysis, no matter how precise it is. For AASR, it is within 1%, and for SCCP, GVN and CP it is within 3% of the most precise alias analysis. It is least effective on LICM, but still within 9% of the effectiveness of the best possible analysis.

Stensgaard and Shapiro-Horwitz (SH-4): Steensgaard and Shapiro-Horwitz (SH-4) perform essentially the same on all the optimizations. For LU, GVR, AASR, and SR, they are within 0.1% of the most precise analysis. For GVN, they are within 1%, for CP, within 1.5%, and for SCCP, within 3% of the best possible analysis. Again, they are least effective on LICM, but still within 6% of the most precise alias analysis.

LU: Very little opportunity (only 0.1%) is left for improving LU beyond what AT, ST, or SH-4 already achieve. We find that for each program, exactly the same number of

loops were unrolled when using AT, ST and SH-4. LU unrolled a few more loops when “no aliases” (NA) is assumed on only 3 programs (175.vpr, 300.twolf and 099.go).

GVR: GVR behaves exactly like LU.

SCCP: ST and SH-4 already achieve about 97% of what NA achieves. In our experiments, ST and SH-4 behave identically with respect to SCCP on all programs. For only 4 programs (loader, simulator, flex and 175.vpr), ST and SH-4 trigger more constant propagation than AT does. However, for about 40% (14 out of 36) of the programs, there is still a little room to improve SCCP.

AASR: The room for improvement is less than 1% and is, in fact, about 0.1% with respect to ST and SH-4. For each program, AASR behaves identically with respect to ST and SH-4 and for only 3 programs (04.bisect, simulator, and 175.vpr) does applying ST or SH-4 instead of AT have any positive impact. NA improves over ST and SH-4 on ASSR for only 2 programs (04.bisect and simulator). However, a sequence anomaly occurs for 099.go with NA.

LICM: The room for improvement is about 9% with respect to AT and 6% with respect to ST and SH-4. However, the large improvement opportunity with respect to AT is slightly misleading because for half of the programs (3 out of 6) on which LICM improves when assuming NA instead of AT, the number of expressions LICM removes is quite low (AT vs. NA: 1 vs. 3 for 01.qbsort, 2 vs. 6 for loader, 1 vs. 2 for 181.mcf). For LICM, ST and SH-4 behave identically for all programs and only 2 programs (compress and simulator) improve using AT. However, NA again produces a sequence anomaly for 099.go.

GVN: The room for improvement is about 2% with respect to AT and about 1% with respect to ST and SH-4. Again, ST and SH-4 behave identically on all programs. For 15 programs ST and SH-4 improve over AT, and for 14 programs NA improves over ST and SH-4. NA, ST, and SH-4 produce sequence anomalies for 099.go and 04.bisect when compared to AT.

CP: The room for improvement is slightly more than 2% with respect to AT and about 1.5% with respect to ST and SH-4. For 3 programs (129.compress, simulator and 175.vpr) ST improves over AT, for one program (197.parser) SH-4 improves over ST and for 11 programs NA improves over SH-4. For CP, five sequence anomalies occur. For 099.go, AT is the most effective alias analysis for CP and NA is the least effective one. For 197.parser, SH-4 is the most effective analysis and both ST and NA are the least effective. For 300.twolf, AT is the most effective analysis and the rest are identical to each other. A similar pattern occurred with ST on 256.bzip2. For 186.crafty, AT was better than ST and SH-4, but using NA propagates the largest number of copies.

SR: Alias analysis has very little impact on SR and the only room for improvement, if any, is around 0.1%. NA slightly influences only 4 programs (256.bzip2, 197.parser, 175.vpr and 099.go), but none of AT, ST, and SH-4 provide a trend.

UCR: Alias analysis influences UCR in a very complex fashion because the resulting SSA graph and changes made by other optimizations both add and remove copies. In these experiments, a more precise alias analysis creates fewer useless copy statements than a less precise one.

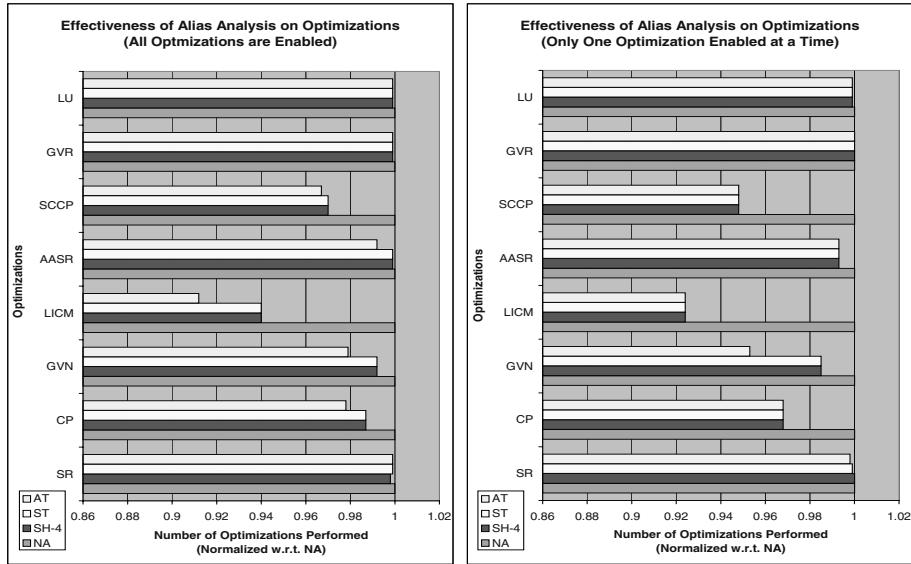


Fig. 2. Effectiveness of alias analysis on optimizations (all optimizations enabled).

Fig. 3. Effectiveness of alias analysis on optimizations (one optimization enabled at a time).

7.4 Optimizations Enabled Independently

Table 6 summarizes the results obtained by performing each optimization independently. This table is structured almost identically to Table 5. It does not include UCR since Scale UCR is only applicable after other optimizations. Fig. 3 graphs this data. The last column of Table 6 indicates that optimization opportunities are, in general, greatly reduced when the compiler applies each optimization independently. This reduction is reflected in higher percentages when the alias analysis does influence individual optimization opportunities in isolation. For all optimizations except Scalar Replacement, ST and SH-4 behave identically on all programs. We first summarize the trends, and then examine each optimization.

Address-taken: For GVR, address-taken analysis is as good as any alias analysis. For LU and SR, its effectiveness is within 0.2%, for AASR, within 1%, and for SCCP, GVN, and CP, within 5% of that of the most precise analysis. For LICM, it is within 8% of the best possible analysis.

Steensgaard and Shapiro-Horwitz (SH-4): Steensgaard and Shapiro-Horwitz (SH-4) essentially have the same effect on all these optimizations. For GVR, they are as good as the most precise analysis. For LU and SR, they are within 0.1%, for AASR, within 1%, for GVN, within 1.5%, and for SCCP and CP, within 5% of the effectiveness of the best possible alias analysis. Again, they are least effective on LICM, but still within 8% of the best possible analysis.

LU: This optimization behaves exactly in the same way in all aspects when the compiler enables all optimizations. This similarity results because LU is the first optimization in the sequence of optimizations applied on programs in the previous set of experiments.

GVR: Alias analysis precision does not have any effect.

SCCP: For SCCP, the improvement opportunity is slightly more than 5%. For 13 programs, NA has a more positive impact on SCCP compared to ST and SH-4. ST and SH-4 provide a benefit to SCCP compared to AT for only one program (175.vpr).

AASR: AT, ST, and SH-4 behave identically for every program. For only 2 programs (04.bisect and simulator) further improvements (less than 1%) are possible.

LICM: About 8% improvement is possible for LICM. For each program, AT, ST, and SH-4 identically influence LICM. NA exposed more optimization opportunities than did AT, ST or SH-4 for 7 programs.

GVN: The margin for improvement is about 5% with respect to AT and about 1.5% with respect to ST and SH-4. For 10 programs, ST and SH-4 proved more effective than AT and for 13 programs, NA exposed more optimization opportunities than did ST or SH-4.

CP: The room for improvement is slightly more than 3%. For each program, the impact of AT, ST and SH-4 on CP were identical. For 12 programs, there is a gap between ST/SW-4 and NA.

SR: Alias analysis precision has very little effect.

8 Conclusion

The “assume no aliases” methodology provides upper bound analysis that is surprisingly tight for scalar optimizations and easy to implement. Our results are for the domain of scalar optimizations, and show that there is little room to improve scalar optimization by improving alias analysis. For this client, a fast and less precise analysis is good enough. Within other domains, such as parallelization, error detection, and memory leak detection, precise pointer disambiguation is more often required for correctness or critical for good performance [34,18,19]. However, by studying the upper bound for these clients, researchers can explore the limits of alias analysis. When the bound and analysis match, there is no need to test more precise analyses and the methodology obviates an entire class of iterative testing. When they do not match, the mismatch can reveal where and how to apply more precise and costly analysis.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
2. T. Austin. Pointer-intensive benchmark suite, version 1.1.
<http://www.cs.wisc.edu/~austin/ptr-dist.html>, 1995.
3. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *ACM Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, San Jose, CA, Oct. 1996.
4. P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software—Practice and Experience*, 27(6):701–724, June 1997.

5. J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages*, pages 232–245, Charleston, SC, Jan. 1993.
6. F. Chow, S. Chan, R. Kennedy, S. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 273–286, Las Vegas, NV, June 1997.
7. F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In T. Gyimothy, editor, *International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267, Linkoping, Sweden, Apr. 1996. Springer-Verlag.
8. R. A. Chowdhury, P. Djeu, B. Cahoon, J. H. Burrill, and K. S. McKinley. The limits of alias analysis for scalar optimizations. Technical Report TR-03-059, University of Texas at Austin, Oct. 2003.
9. K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Proceedings of the 16th Annual ACM Symposium on the Principles of Programming Languages*, pages 49–59, 1989.
10. K. D. Cooper and J. Lu. Register promotion in C programs. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 308–319, Las Vegas, NV, June 1997.
11. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th Annual ACM Symposium on the Principles of Programming Languages*, pages 25–35, Austin, TX, Jan. 1989.
12. M. Das, B. Liblit, M. Fahndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *The 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 260–278, Paris, France, July 2001. Springer-Verlag.
13. A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, Montreal, June 1998.
14. A. Diwan, K. S. McKinley, and J. E. B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23(1):30–72, Jan. 2001.
15. A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and effective analysis of statically-typed object-oriented languages. In *ACM Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 344–355, San Jose, CA, Oct. 1996.
16. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.
17. R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, Jan. 1998.
18. S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *International Static Analysis Symposium*, pages 214–236, San Diego, CA, June 2003.
19. D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 168–181, San Diego, CA, June 2003.
20. M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analysis. In *The 5th International Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 57–81, Pisa, Italy, Sept. 1998. Springer-Verlag.
21. M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 112–123, Portland, OR, Aug 2000.

22. W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. *ACM SIGPLAN Notices*, 28(6):56–67, 1993.
23. D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 199–215, Toulouse, France, Sept. 1999.
24. K. S. McKinley, J. Burrill, B. Cahoon, J. E. B. Moss, Z. Wang, and C. Weems. The Scale compiler. Technical report, University of Massachusetts, 2001. <http://ali-www.cs.umass.edu/~scale/>.
25. E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, CA, June 1995.
26. Rutgers. PROLANGS benchmark suite, data programs. <http://www.prolangs.rutgers.edu/public.html>, 1999.
27. M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In P. V. Hentenryck, editor, *Lecture Notes in Computer Science*, 1302, pages 16–34. Springer-Verlag, 1997. Proceedings from the 4th International Static Analysis Symposium.
28. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 1–14, Paris, France, Jan. 1997.
29. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 21–24, St. Petersburg, FL, Jan. 1996.
30. P. A. Stocks, B. G. Ryder, W. Landi, and S. Zhang. Comparing flow and context sensitivity on the modifications-side-effects problem. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 21–31, Clearwater, FL, Mar. 1998.
31. O. Traub, G. Haolloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 142–151, Montreal, June 1998.
32. Z. Wang, D. Burger, K. S. McKinley, S. Reinhardt, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 388–398, San Diego, CA, June 2003.
33. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.
34. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 1995.
35. S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 91–103, Atlanta, GA, June 1999.

Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs*

Venkatesh Prasad Ranganath and John Hatcliff

Department of Computing and Information Sciences, Kansas State University,
234 Nichols Hall, Manhattan KS, 66506, USA
`{rvprasad,hatcliff}@cis.ksu.edu`

Abstract. In this paper, we show how previous work on escape analysis can be adapted and extended to yield a static analysis that is efficient yet effective for reducing the number of *interference dependence* edges considered while slicing concurrent Java programs. The key idea is to statically detect situations where run-time heap objects are reachable from a single thread and use it to prune spurious interference dependence edges. We also show how this analysis can be extended to reduce the number of *ready dependence* edges – dependences that capture indefinite delay due to Java synchronization constructs.

The analysis we describe has been implemented in the Bandera[9] slicer which is being applied to reduce the size of software models for model-checking. Using this implementation, we give experimental results that demonstrate the effectiveness of our approach. We believe leveraging escape information in the manner we describe is a crucial element in scaling slicing techniques to larger concurrent object-oriented programs.

1 Introduction

Program slicing has proven to be effective for program debugging, understanding, and specialization. Despite the widespread use of concurrent object-oriented languages such as Java and C#, the amount of work on relevant techniques for slicing programs in these languages is much less than the amount of work devoted to slicing sequential programs written in languages such as C. Most of the algorithms [32] proposed for slicing concurrent programs extend the structures and algorithms proposed for slicing sequential programs. Results from such extensions are often overly pessimistic as they adapt a rather simplified approach to deal with the additional dependencies ([13]) that arise in concurrent programs.

Interference dependence [18] is one such additional dependence that is similar to the notion of *data dependence* that exists in conventional slicing for sequential programs. In a sequential program, a statement s_2 is said to be *data dependent* on another statement s_1 if a definition to (i.e., an assignment to) variable x at s_1 reaches a use of x in s_2 along a particular program control-flow path. In concurrent program slicing, an interference dependence may be explained intuitively as

* This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), and by Rockwell-Collins and by Intel Corporation (Grant 11462).

a data dependence between s_1 and s_2 where these statements occur in different threads, i.e., thread interleaving allows a definition of x at s_1 to be used at s_2 . Just as with data dependences, calculation of interference dependences becomes complicated when reaching definitions associated with heap-allocated objects are considered. In particular, precise alias information is required to avoid retreating back to safe but very conservative definitions such as “any definition of a field f of any object o_1 of type T can reach any use of field f of any object o_2 of the same type T ”.

Efficient calculation of precise interference information for concurrent OO programs is challenging because it is difficult to statically reason about all the possible run-time thread interleavings. Existing approaches for Java slicing tend to fall into two categories: approaches that use exponential symbolic execution to approximate interleavings [5,18,24] and approaches that avoid the high cost of the symbolic execution technique by making very conservative assumptions [13,32].

The techniques that we propose in this paper are based on two well-studied forms of static analysis for concurrent Java – escape analysis and race detection analysis – that can be leveraged to improve the precision of interference dependence, hence, the resulting program slice. Our approach is much more precise than the conservative approaches mentioned above and much less expensive than the symbolic execution-based techniques.

Until now, as far as we know, there has been no work reporting the use of these analyses to improve the precision of dependence information required to slice concurrent OO programs. In fact, the recent work by Krinke ([19]) that uses symbolic execution to prune interference edges does not consider dynamically created objects or pointers, and so our approach can also be viewed as complementary to such techniques.

The contributions of this paper are as follows.

- We adapt and extend the escape analysis by Ruf [28] (which was targeted to removal of unnecessary synchronization) to yield more precise information about sharing and aliasing of objects across threads by changing the points at which flow data is merged and by adding additional symbolic entities to track specific def/use and wait/notify relationships.
- We describe how to apply the results of escape analysis to improve the precision of calculation of interference and ready dependences in the context of program slicing.
- We report experimental results from applying our techniques to Java Grande benchmarks that demonstrate their effectiveness in removing spurious dependence edges.

The following section defines the notions of interference and ready dependence used in slicing concurrent programs, and it presents an example to illustrate the reductions that can be achieved by using escape analysis. Section 3 presents our modified version of Ruf’s escape analysis. Section 4 explains how the information from our analysis can be used to drive transformations such as

program slicing. Section 5 presents the results of our experiments. Section 6 describes related work, and Section 7 concludes and describes possible extensions to our approach.

2 Background

2.1 Data Flow across Threads in Java

Interference dependences can only arise when an update is made to a data item that is “shared” between two or more threads. Thus, we begin by considering how an object can be shared between threads using the simple program of Example 1 (ignore the `mainAlt` method for now). Java[12] supports threads via `Thread` objects.¹ The example program has three threads: the main thread which runs the method `main` in class `Home`, a thread associated with an instance of `Man`, and a thread associated with an instance of `Wife`. In Java, a thread becomes “alive” when `java.lang.Thread.start()` is dispatched on the associated thread object (e.g., the calls to `start` at line 30 and line 31). This dispatch results in the invocation of `run()` on the receiver of the `start()` method (e.g., the call to `start()` at line 31 causes the newly activated thread to invoke the `run` method at line 16).

Let us now consider all possible ways in which a heap object o created by a thread t can be communicated to a newly created thread t' . Java disallows any parameters for the `run()` or `start()` methods of `java.lang.Thread`. Thus, the only way to provide data to the `run` method of t' is to have it be reachable² from the `this` variable of the `Thread` instance for t' or to have it be reachable from static fields (in essence, global variables). Note also that there are only two ways to assign data/values to instance fields of an object. One is via direct assignments to reachable fields. The other is via invocation of an instance method in which the fields reachable via the receiver will be assigned data reachable from the arguments to the method. For instance, in the example program, the `main` thread communicates the savings account object to the `run()` method of the `Wife` thread by invoking the constructor of `Wife` with the account as a parameter; the constructor then assigns the account object to the `savings` field reachable from `this`. The main properties of this example that we will use to illustrate object escape properties are: (a) an escaping savings account object ends up being shared between `Man` and `Wife` instances and (b) a new `Acct` object is created and used only in the `run()` method for `Wife`.

2.2 Interference Dependence

Definition 1. Let P be a program, o be an object with field f , and t_1 and t_2 be threads such that $t_1 \neq t_2$. If there exists an execution trace of P such that

¹ The term *thread* indicates an executing entity and the term *thread object* indicates an instance of `java.lang.Thread` or any of its subclasses.

² From here on, “reachable” will signify reachability based on following references held in accessible fields.

```

1  class Acct {
2    protected int balance;
3    Acct() {
4      balance = 100;
5    }
6  } // End of class Acct
7
8  class Man extends Thread {
9    protected Acct savings;
10   protected Acct checking;
11   Man(Acct act) {
12     this.savings = act;
13     checking = new Acct();
14     int i = checking.balance;
15   }
16  public void run() {
17    savings.balance += 20;
18    synchronized(savings) {
19      savings.notify();
20    }
21    checking.balance += 10;
22  }
23 } // End of class Man
24
25 public class Home {
26  public static void main(String[] s) {
27    Acct savings = new Acct();
28    Wife wife = new Wife(savings);
29    Man man = new Man(savings);
30    wife.start();
31    man.start();
32  }
33
34  public static void mainAlt(String[] s) {
35    Acct savings = new Acct();
36    Thread wife, man;
37    for(int i = 2; i <= 3; i++) {
38      if (i % 2 == 0)
39        wife = new Wife(savings);
40      if (i % 3 == 0)
41        man = new Man(savings);
42    wife.start();
43    man.start();
44  }
45 } // End of class Home
46
47 class Wife extends Thread {
48  protected Acct savings;
49  protected Acct checking;
50  Wife(Acct act) {
51    this.savings = act;
52    checking = new Acct();
53  }
54  public void run() {
55    Acct newAcct = new Acct();
56    synchronized(savings) {
57      savings.wait();
58    }
59    savings.balance -= 20;
60    synchronized(checking) {
61      checking.wait();
62    }
63    checking.balance -= 10;
64    newAcct.balance += 10;
65  }
66 } // End of class Wife

```

Fig. 1. A simple Java program illustrating object sharing between threads.

f is written at trace state s_m by a statement at program point m executed by t_1 and read at state s_n by a statement at program point n executed by t_2 (with s_n occurring after s_m) and no write to o.f occurs between s_m and s_n , then n is interference dependent on m.

Given the above definition of interference dependence in terms of traces of concurrent programs, it is obvious that precise and efficient calculation of interference edges is infeasible. Therefore, one must generally fall back to safe approximations that are cheaper to compute such as the following:

If a field f of object o is being written and read at program points m and n, respectively, and m and n occur in different threads t_m and t_n , then n is interference dependent on m

By referring to the example of Figure 1, we now explain (1) how interference edges are detected in previous presentations of slicing [13,32], (2) how our modification of Ruf's escape analysis can be used to safely reduce the number of edges, and (3) how simple extensions to track aliasing gives even further reduction.

Type-based approach: A simple approach is to classify field read/writes as interfering if m represents a read of an expression such as $a1.f$, n is a write

to `a2.f`, and `a1.f` and `a2.f` have identical signature (i.e., both instances of `f` represent the same field of the same class – note that field resolution in Java is based on the static type of the primary³ and **not** its dynamic type) then `m` is interference dependent on `n`. While this approximation of interference dependence is easy to compute, it includes inference edges even for objects that are not shared. Hence, it leads to spurious interference dependence edges such as ones between line 21 and both line 59 and line 64.

Leveraging escape analysis: The definition of interference dependence implies that objects that induce interference dependences must be accessible from more than one thread at run-time. Escape analyses (e.g., Ruf’s) can determine when objects are accessed by at most one thread, and these objects cannot participate in an interference dependence. In Figure 1, typical escape analyses will mark `newAcct` in `Wife.run()` as holding only thread-local objects. This allows the elimination of a spurious edge between line 21 and line 64 generated in the naive method. However, Ruf’s escape analysis would mark `checking` and `savings` in `Wife.run()` as escaping since they are both accessed in the constructor for `Wife` which is executed by the `main` thread – *not* the `wife` thread. This means that the edge between line 21 and line 59 cannot be removed.

Leveraging alias information: Alias information can be used to prune edges relating `checking.balance` and `saving.balance`. For example, the knowledge that the primaries in line 21 and line 59 are not aliases can be used to remove the interference edges between these lines. One of our several modifications that we give to Ruf’s analysis is an extension that uses what we call *share entities* to detect such information (see Section 4.1).

2.3 Ready Dependence

In simple words, a statement `m` is ready-dependent on a statement `n` if the execution of `m` can be infinitely delayed due to the fact that `n` fails to complete its execution and `n` is a synchronization related construct. This notion of dependence is relevant when using slicing to generate reduced models for checking temporal properties (specifically, liveness properties) [13]. We shall use `notify()` to denote both `notify()` and `notifyAll()` methods in `java.lang.Object` and `wait()` to denote all overloaded versions of `wait()` in `java.lang.Object`.

Definition 2. *If `m` and `n` are program points in a given Java program then `m` is ready dependent on `n` if any one of the following is true.*

1. *`m` and `n` occur in the same thread and `m` is reachable from `n` and `n` is an **entermonitor**.*
2. *`m` and `n` occur in the same thread and `m` is reachable from `n` and `n` is an invocation of `wait()`.*
3. *`m` and `n` occur in different threads and `m` is an **entermonitor** statement on object `o` and `n` is an **exitmonitor** statement on object `o`.*

³ In `a.f` and `a[b]`, `a` is referred to as the primary.

4. *m and n occur in different threads and n is an invocation of **notify()** on object o and m is an invocation of **wait** on object o.*

Condition 1 and 2 are uninteresting for us as they can be dealt with a simple sequential control flow graph. Condition 3 and 4 provide interesting cases as they span across threads. From here on, by 'ready dependence' we mean 'ready dependence resulting from conditions 3 and 4 only'. The techniques we consider for pruning interference dependences can also be used to prune ready dependence (e.g., if an object *o* is not shared then it cannot generate a ready dependence via conditions 3 or 4).

In Figure 1, a naive type-based ready dependence calculation as implemented previously in Bandera[9] will report ready dependence edges between line 19 and both line 57 and line 61. However, the latter edge is a spurious one which cannot be eliminated by escape information. However, it can be eliminated using alias information computed using what we call *ready entities* as described in Section 3.

3 Details of the Analysis

Although there are numerous escape analyses, we choose to work with Ruf's escape analysis because it is relatively easy to understand, straightforward to implement, scales well, and its particular strategy for identifying thread-local data yields more precise results than several other approaches (see [28] for details).

Our goal is to create an analysis that calculates information that will enable the following questions to be answered.

- Given a field/array read expression *m* and a field/array write expression *n*, is *m* interference dependent on *n*?
- Given a **entermonitor** statement *m* and a **exitmonitor** statement *n*, is *m* ready dependent on *n* under condition 3 in the definition of ready dependence?
- Given a **wait()** call-site *m* and a **notify()** call site *n*, is *m* ready dependent on *n* under condition 4 in the definition of ready dependence?

The analysis proceeds in three phases. The first phase collecting information about the system to be used during phase two and three. In phase two, intra-procedural analysis happens along with bottom-up(in the call graph) interprocedural analysis in an interleaved fashion. In phase three, the information from the callers is propagated to the callees. After presenting the details of the analysis, we summarize how it differs from Ruf's original version.

3.1 Alias Sets and Contexts

Each program variable *v* is associated with an *alias set* – an abstract object that summarizes properties of concrete objects that may be referenced by *v* at

run-time. An alias set for v is either \perp (indicating that only null reference values are assigned to v at run-time, or that v is of non-reference type) or a tuple of properties as follows

$\text{aliasSet} ::= \perp \mid \langle \text{global}, \text{fieldMap}, \text{accessed}, \text{escapes}, \text{waits}, \text{notifies}, \text{readyEntities} \rangle.$

The elements of the tuples are described below.

global is a boolean that when $true$ indicates that the abstract object is reachable via static fields.

fieldMap associates each field f of the abstract object with an alias set abstracting the concrete objects that may be referenced by f at run-time, i.e., it maps fully quantified field names to alias sets. $\$ELT$ is a special field used to represent all cells in a dimension of an array.

accessed is a boolean that when $true$ indicates that the abstract object was read or written.

escapes is a boolean that when $true$ indicates that the abstract object is accessed by more than one thread.

waits is a boolean that when $true$ indicates that the abstract object received a `wait()`.

notifies is a boolean value that when $true$ indicates that the abstract object received a `notify()` or `notifyAll()`.

readyEntities is a set of entities drawn from a domain E . It forms a finer partition than the alias sets. This entity set indicates the participation of the objects represented by the alias set in wait/notify relationship (leading to a ready dependence). Specifically, if the intersection of the ready entities for some alias sets a_1 bound to variable v_1 and a_2 bound to v_2 is non-empty, then invocations of wait/notify on v_1 and v_2 can lead to a ready dependence.

As the analysis manipulates alias sets, certain invariants representing consistency properties must be enforced. If an alias set has $\text{global}=true$, then all alias sets reachable via fieldMap should have $\text{global}=true$ (indicating that if an object o is reachable from more than one thread, then the objects referenced by o 's fields are as well). If $\text{global} = true$, then $\text{escapes} = true$. Finally, if $\text{readyEntities} \neq \emptyset$ then $\text{escapes} = true$.

Upon creation of an alias set, all boolean elements of the alias set are set to $false$ except for accessed which is set to $true$ ⁴ readyEntities is set to \emptyset and the fieldMap is empty. Alias sets of fields are created on demand. Alias sets are operated on by `clone`, `unify`, and `unifyAtStart` operations.

The `clone` operation creates a new alias set isomorphic to the existing one. As a space optimization, when cloning alias sets marked as global, a reference to the existing alias set is returned. In addition, the elements of readyEntities are always shared when cloning.

⁴ The mere existence of the alias set can be substituted for the truth value of accessed . Hence, this field is redundant, however, it is used as it makes explanation of the analysis more intuitive.

Domain

$v \in V$ set of variables
 $f \in F$ set of fields
 $m, p \in M$ set of methods
 $a, r, e \in A$ set of alias sets
 $mc, sc \in C$ set of alias contexts
 $s \in CS$ set of call sites

Mappings

$AS : V \rightarrow A$ alias set lookup
 $MC : M \rightarrow C$ method context lookup
 $CALLEES : M \times V \rightarrow \wp(M)$ callee lookup
 $SCC : M \times \wp(M)$ SCC lookup
 $MULTI_EXEC : CS \rightarrow \{true, false\}$ multiply executed call-site lookup

Rules

statement	action
$v_1 = (t)v_2$	$unify(AS(v_1), AS(v_2))$
$v_1 = v_2.f$	$unify(AS(v_1), AS(v_2).fieldMap(f))$
$v_2.f = v_1$	
$v_1[] = v_2$	$unify(AS(v_1).fieldMap(\$ELT), AS(v_2))$
$v = v_1[]$	
$return v$	$unify(AS(v), AS(r))$
$throw t$	$unify(AS(t), AS(e))$
$v_r = v_t.m(a_1, \dots, a_n)$ let $sc = \langle AS(v_t), \langle AS(a_1), \dots, AS(a_n) \rangle, AS(v_r), e \rangle$	$\forall p \in CALLEES(m, v_t).$ $let mc = MC(p) and unifier = unify$ $if m = java.lang.Thread.start then$ $unifier = unifyStart$ $if SCC(m) = SCC(p) then$ $unifier(sc, mc)$ $else$ $unifier(sc, newInstance(mc))$ $if MULTI_EXEC(v_t.m(a_1, \dots, a_n)) and$ $m = java.lang.Thread.start then$ $unifier(sc, sc)$

Fig. 2. Domains, mappings and rules used in intra-procedural analysis. In the rules, m represents the method in which the rules are being applied and e is the alias set corresponding to the exceptions thrown in m .

The *unify* operation merges the information represented by elements of the tuple (except for *escapes* and *readyEntities* as explained below). The unification of the boolean and map values is defined as the join under the boolean lattice (with *true* as the top element) and function lattice, respectively. The domain of the resulting *fieldMaps* is the union of the domains of the maps being unified. The alias sets of the field names occurring in the domains of both the maps are (recursively) unified, as well.

The *escapes* and *readyEntities* elements represent information that spans threads, and thus these elements should only be merged at thread start sites. The *unifyAtStart* operation implements this distinct form of merging. Intuitively, an object should be considered escaping when it is accessed in more than one thread. To capture this, the unification of the *escapes* element is defined as the join under the boolean lattice when either one of the arguments is *true* (i.e., if objects are

Table 1. Rules to unify *escapes* under *unifyAtStart* operation when **arg1.escapes = arg2.escapes = false**.

arg1.accessed	arg2.accessed	result.escapes
false	false	false
false	true	false
true	false	false
true	true	true

already marked escaping in at least one of the alias sets being merged, then they should continue to be considered as escaping). However, when the *escapes* field in both the alias sets being unified is *false*, the value of *escapes* in the merged alias set is given by Table 1 (intuitively, the arguments to *unifyAtStart* are alias sets coming from two different threads, and the resulting merged alias set should only be marked as escaping (i.e., shared) if *accessed* is true from both threads).

The result of unifying *readyEntities* is determined by the *waits* and *notifies* elements of the alias sets being unified. The *readyEntities* of the alias sets being unified is modified only when *waits* element of one alias set is *true* and the *notifies* element of the other alias set is *true*. In this case, the *readyEntities* set is modified by injecting a fresh entity into the *readyEntities* of the alias set occurring in the site context (introduced below). In all other cases, *readyEntities* set is unmodified in the result of unification.

An *alias context* is an aggregate data structure used to represent the flow of information into and out of a method. An alias context is a tuple of alias sets corresponding to **this**, method arguments (a_1, a_2, a_3, \dots), return value(r), and the exceptions thrown by the method (e).

$$\text{aliasContext} ::= \langle \text{this}, \langle a_1, a_2, a_3, \dots \rangle, r, e \rangle$$

Like alias sets, alias contexts support *clone* and *unify* operations. *clone* creates a new alias context isomorphic to the existing one. The *unify* operation is a point-wise extension of alias set unification to alias contexts tuples. For the sake of simplicity, we use the term *method context* to indicate the alias context of a method at its entry point and the term *site context* to indicate the alias context at a call-site.

Both alias set and alias context are implemented as union-find data structures [29].

3.2 Phase One

The analysis begins by forming abstract representatives for groups of one or more concrete threads. The abstract representative of a group of threads is a call graph rooted at the start site of each thread. Thus, there is an abstract thread for each start site, and this abstract thread represents all concrete threads started at that site. The complete collection of such call graphs for a program is called a *threaded call graph*.

The threaded call-graph is constructed as follows. First, a value-flow-based framework is used to construct a traditional call graph [27]. Strongly-connected components (SCC) of the call graph are calculated. This is followed by identifying call-sites which invoke `java.lang.Thread.start()`. If any of these call-sites can be executed more than once during the life-time of the system, (i.e., if they occur in a method called more than once, if the method occurs in a SCC of size greater than one, or the call-site is enclosed in a loop), then the associated call-graph is marked as possibly representing more than one concrete thread. The intuition is that if an object is thread-local to an abstract thread representing multiple concrete threads then one must assume that it can be shared between multiple concrete threads created at the same start site – the analysis cannot distinguish between threads started at the same `start`-call site.

3.3 Phase Two

This phase starts off by creating alias sets with *global=true* for static fields. It then performs an interprocedural analysis by processing each method in each SCC in a bottom-up fashion on the call graph. The processing of the methods includes the creating of the method context and performing flow-insensitive intra-procedural analysis on the method as described below. The effect of this phase is to accumulate information about object accesses in the call-graph of a particular thread t , bubble this information up, and expose it at the start call site for a particular thread.

Intra-procedural Analysis. Figure 2 presents the rules for processing the statements of a method during intra-procedural analysis. These rules ensure that the aliasing of data inside the method is represented appropriately in method contexts and global alias sets. In a method, all statements not containing calls to `start()` are processed before processing statements containing calls to `start()` to capture the effect of `post-start()` site data accesses at `start()` sites. During processing, if an alias set for a variable does not exist, a new instance is created.

The interesting rule is the one for method invocation. At a call-site, if the caller and the callee exist in the different strongly connected components in the call graph then the method context of the callee is cloned and the clone is unified with the site context to achieve context sensitivity. When both caller and callee occur in the same SCC, the method context of the callee is unified with the site context to achieve an effect similar to simple fixed-point iteration.

The rule for method invocation varies the alias set unifier depending on the method being invoked. For method calls other than `start()`, the same thread executes the caller and the callee (thus *unify* is used as the unifier operation). For `start()`, different threads execute the caller and the callee, and thus *escapes* in the alias sets needs to be updated (thus *unifyAtStart* is used as the unifier operation). As *readyEntities* depend on *waits* and *notifies* of alias sets representing values in different threads, they are also updated via *unifyAtStart*. If the call-site invokes `start()` and is determined to be executed multiple times,

the site context is unified with itself via *unifyStart* to account for the effect of multiple executions as described in Section 3.7.

3.4 Phase Three

In this phase, the call hierarchy is traversed in a top-down fashion. This ensures that any changes to the information accumulated by the parent thread are exposed to children threads.

Call-sites are the only points of processing in this phase. At each call-site encountered in the top-down traversal of the call graph, each alias set in each method context is visited recursively. On visiting an alias set of a method context, the *escapes* element of the alias set is joined with the *escapes* element of the alias set's counterpart in the site context. As for *readyEntities*, the values are not joined (which would destroy context sensitivity), but rather values from the site context alias sets are injected into the *readyEntities* of method context alias sets.

3.5 A Comparison to Ruf’s Analysis

We now summarize how our escape analysis differs from the original version given by Ruf.

- Ruf’s analysis is geared towards the removal of unnecessary synchronization operations. Hence, Ruf’s alias sets have elements *synchronized* and *syncThreads* to capture information indicating whether or not objects represented by the alias set are used in synchronization and the number of threads that synchronize on the objects, respectively. However, the alias sets in our analysis have other fields tailored to calculate escape information and dependence information.
- Unlike Ruf’s analysis, we have used two versions of unification (*unify* and *unifyAtStart*), and the form that is used at `start()` call-sites alters the escape information while the other form does not.
- The unification rules in Ruf’s analysis mark an alias set as synchronized only at synchronization expressions. However, in our analysis, if a variable occurs in an access expression then the associated alias set is marked as accessed and the decision to mark the alias set as escaping is deferred to our new *unifyAtStart* unification operation that is used at `start()` call-sites.

3.6 Example

When the analysis is applied to the program of Figure 1, Phase 1 is uninteresting since there aren’t any `start()` call sites that will be executed multiple times in this simple program.

Phase 2 starts by processing `Acct.Acct()`, `Wife.Wife(Acct)`, `Man.Man(Acct)`, `Wife.run()`, `Man.run()`, and `Home.main()`. On processing `Acct.Acct()`, a method context $\langle \alpha_0, \langle \rangle, \alpha_0, \perp \rangle^5$ is created where and

⁵ The alias set, α_0 , indicates that the constructor returns a value to signify the creation of the object and the invocation of the constructor on it.

$\alpha_0 = \langle \text{false}, \langle \rangle, \text{false}, \text{false}, \text{false}, \emptyset \rangle$ where α_0 is a newly created alias set⁶. Note that the *accessed* element of α_0 is set to *false* as a result of WHB optimization.

The method context at the start of the processing of `Wife.Wife(Acct)` will be $\langle \alpha_1, \langle \alpha_2 \rangle, \alpha_1, \perp \rangle$ in which α_1 and α_2 are newly created alias sets. After processing the method, $\alpha_1 = \langle \text{false}, \{\text{savings} \rightarrow \alpha_2, \text{checking} \rightarrow \alpha_3\}, \text{false}, \text{false}, \text{false}, \emptyset \rangle$ where α_3 is a newly created alias set and $\alpha_2.\text{accessed} = \text{false}$. A similar method context $\langle \alpha_4, \langle \alpha_5 \rangle, \alpha_4, \perp \rangle$ occurs for `Man.Man(Acct)` where $\alpha_4 = \langle \text{false}, \{\text{savings} \rightarrow \alpha_5, \text{checking} \rightarrow \alpha_6\}, \text{false}, \text{false}, \text{false}, \emptyset \rangle$.

Upon processing `Wife.run()`, its method context will be $\langle \alpha_7, \langle \rangle, \perp, \perp \rangle$ where $\alpha_7 = \langle \text{false}, \{\text{savings} \rightarrow \alpha_8, \text{checking} \rightarrow \alpha_9\}, \text{true}, \text{false}, \text{false}, \emptyset \rangle$, $\alpha_{8/9} = \langle \text{false}, \langle \rangle, \text{true}, \text{false}, \text{true}, \emptyset \rangle$. The local variable `newAcct` in `Wife.run()` is associated with α_{10} , however, this alias set is not visible from the method context. We omit the details of these alias sets from here on due to space constraints. A similar method context occurs for `Man.run()` except that the *notifies* element in the alias sets of `savings` and `checking` fields is set to `true` and `false`, respectively.

The processing of the line 27 creates a new alias set, α_{10} , associated with the local variable `savings`. The processing of line 28 associates `wife` with $\alpha_{11} = \langle \text{false}, \{\text{savings} \rightarrow \alpha_{10}, \text{checkings} \rightarrow \alpha_{12}\}, \text{true}, \text{false}, \text{false}, \emptyset \rangle$. The processing of line 30 will unify $\langle \alpha_{11}, \langle \rangle, \perp, \perp \rangle$ with an isomorphic copy of the method context of `Wife.run()`. This will result in *accessed* and *wait* in α_{10} being set to `true` due to unification with an isomorphic copy of α_8 . By the unification rules in Table 1, *escapes* element of α_{10} is set to `true`.

The processing of line 31 will unify α_{10} with an isomorphic copy in which *accessed* is set to `true`, hence, based on the unification rules at `start()` call-sites $\alpha_{10}.\text{escapes}$ is set to `true`. Also, as *notifies* will be `true` on the isomorphic copy, by the unification rule for *readyEntities*, a new object is inserted into $\alpha_{10}.\text{readyEntities}$ indicating that `wait()` and `notify()` on `saving` should be considered for ready dependence.

3.7 Allocation Sites Executed Multiple Times

Call-sites and allocation sites that are executed multiple times pose a major obstacle for static analyses such as escape analysis.

For example, if `Home.mainAlt()` was considered as the entry point to the system, then the abstract thread corresponding to the thread allocation site in the loop at line 37 in Figure 1 represents more than one run-time thread object. In such situations and also when `start()` call-sites are executed multiple times, Ruf's analysis will provide pessimistic information by marking all reachable alias set from such site contexts as shared.

However, upon unrolling the loop once and analyzing using the rules given earlier, each alias set reachable from the site context will be unified with itself.

⁶ We use the term *newly created alias set* to indicate an alias set unaltered after creation.

Hence, in our analysis, we use this observation in Phase 3 to unify the site context with itself via *unifyStart* before propagating the information across a `start()` call-site that is executed multiple times (note that due to the definition of unification on escape and ready information, unifying an alias set with itself does not necessarily correspond to the identity function). This captures the effect of the loop on the flow of information. This is not done at other call-sites as they all occur in the same thread, hence, the optimization cannot affect escape information.

3.8 Complexity

The algorithm visits each node in the call graph twice and at each node it visits the statements of the methods once. The worst case processing time for unification is dictated by the recursive nature of the data structures used in the program, in particular when propagating information across method boundaries. Hence, the worst case time complexity for the analysis should be in the order of $O(N + S)$ where N is the number of nodes in the call graph and S is the total number of statements in the system if time complexity of object construction is considered as a constant.

New instances of alias sets are created at method call sites. Hence, a data structure with m fields being passed across n call sites at each level of a call chain of length p can lead to alias sets whose accumulated size is in the order of $O(m * n^p)$. However, in reality, large data structures are recursive in nature rather than a single big chunk. Hence, reaching worst case space complexity should be a rarity.

4 Transformations

We now describe how the information computed by our escape analysis can be used in slicing and other applications.

Our slicer implementation proceeds in two phases: an initial phase collects various dependence information, and the second phase follows the dependence edges to calculate the program slice. As we have indicated, the results of our analysis can be used to prune interference dependence edges between non-escaping primaries. Also, ready dependence edges can be pruned when the *readyEntities* of the alias sets corresponding to the receivers of `wait()` and `notify()` are disjoint.

An interesting by-product of this analysis is detection of buggy `wait()`s and unnecessary `notify()` calls in the system. From the semantics of `wait()` and `notify()` it can be inferred that the receiver object should be shared across threads for these calls to be safe and useful, respectively. Hence, if *waits* and/or *notifies* element of an alias set are set to *true* and *escapes* is set to *false*, then the corresponding invocations can be concluded as buggy and unnecessary. In the former case, the developer can use this information to make the program safe by avoiding infinite waits. In the latter case, the corresponding invocations of `notify()` can be removed to improve run-time performance.

As with other escape analyses, we can use the result of our analysis to remove unnecessary synchronization and stack allocate objects. Both optimizations can be triggered when the *escapes* element of the alias set associated with the participating variable is set to *false*.

4.1 Increased Precision Using *Share Entities*

Detection of interference based on escape information is still pessimistic as it may be the case that two field access expression may have the same signature but may correspond to different run-time primaries. This can be remedied to some degree by extending the alias set with the following three elements.

read is a boolean that when *true* indicates that the value was read, *written* is a boolean that when *true* indicates that the value was written, and *shareEntities* is a set of object entities that represents objects which participate in the interference dependence. It is similar to *readyEntities* in meaning. From the nature of interference dependence, $\text{shareEntities} \neq \emptyset \Rightarrow \text{escapes} = \text{true}$.

Similar to the unification rule of *readyEntities*, the *shareEntities* of the alias sets being unified is modified only when *read* element of one alias set is *true* and the *written* element of the other alias set is *true*. In this case, the *shareEntities* set is modified by injecting a fresh entity into the *shareEntities* of the alias set occurring in the site context. In all other cases, *shareEntities* set is unmodified in the result of unification.

With this information, access expressions can be considered for interference dependence only when the *shareEntities* of their corresponding alias sets are not disjoint. This information can also be used to improve the precision of ready dependence based on condition three of Definition 2.

5 Experimental Results

We have implemented all variants of the analysis described in the previous sections in the the Indus Slicer component of the Bandera tool set [16]. Soot [30] was used as the intermediate representation. Unlike Ruf's analysis, we use *local variable splitting* transformation of Soot instead of SSA representation to ensure that variables are defined only once.

We summarize the results of applying the analysis to the Java Grande[17] benchmark suite. Java Grande provides three classes of applications: low-level operations, kernels, and large scale applications. In our experiments, we considered each method with the signature `public static void main(String[])` as representing a possible entry point into the system.

Table 5 presents the results of applying the previously described escape analysis followed by the generation of interference dependence edges. We not give timing data in the table, *since all the runs completed in less than 3 seconds* on 2.5GHz machine with 2GBs of RAM using Sun JDK1.4.2 with a maximum heap-size of 1GB. Note that this timing data does not include the cost of parsing, basic control-flow analysis and call-graph construction; the call-graph and

Table 2. Results of analysis and interference dependence generation applied to Java Grande benchmarks.

Program Names	Reachable methods	Interference edges based on		
		Type	Escape info	Entity info
JGFBARRIERBENCH	117	117	29	29
JGFFORKJOINBENCH	121	1929	155	16
JGFSYNCBENCH	87	122	36	36
JGFCRYPTBENCHSIZEA	110	1228	180	58
JGFLUFACTBENCHSIZEA	139	3624	181	24
JGFSORBENCHSIZEA	143	2318	162	23
JGFSERIESBENCHSIZEA	136	2138	157	8
JGFSPARSEMATMULTBENCHSIZEA	144	3833	183	14
JGMOLDYNBENCHSIZEA	146	8026	373	209
JGMONTECARLOBENCHSIZEA	394	5245	309	11
JGFRAYTRACERBENCHSIZEA	157	1330	171	166

threaded-graph construction complete in less than 100 milli-seconds for each example.

The “Type” column in the table contains the baseline data for our experiments – it shows the number of interference edges created based on the simple field signature and value type-compatibility approach described in Section 2.2. The “Escape Info” column shows the number of interference edges detected using escape analysis *without share entities* optimization. These results indicate an order of magnitude reduction of the simple previous approach. The “Entity Info” column shows the number of interference edges detected using escape analysis along *with share entities* optimization of Section 4.1. These results indicate that the simple extension of using entities to form relationships between read/writes can dramatically improve the precision of the analysis with marginal increase in time and space. We believe that this optimization can be incorporated in other variants of escape analysis to obtain more precise results. Given the low cost (time) of the analysis, the results provide convincing evidence that escape information should be leveraged in slicing concurrent object-oriented programs.

6 Related Work

Numerous escape analyses have been proposed to calculate if an object escapes (i.e., can be accessed outside of) a particular method m and/or thread t . Ruf’s equivalence-class-based analysis [28] for calculating if an object is only accessed in a single thread and Choi et.al.’s fixed-point-based analysis [6] to calculate if an object escapes a method or a thread are two well-known escape analyses. Aldrich et.al. [1], Blanchet [2], and Bogda and Hözle [3] also propose similar analyses to improve runtime performance of Java programs by removing unnecessary synchronization and to enable stack allocation of objects.

Similarly, numerous data-race detection algorithms have been proposed. Choi et.al. propose an on-the-fly technique[7] to detect data-race conditions (i.e., situa-

tions where accesses to shared objects are not protected by locks). This requires instrumentation and various optimizations such as static data-race detection analysis as a pre-phase to the dynamic analysis [8]. Flanagan and Freund proposed a annotated type-based technique [10] along with an annotation inference mechanism which is similar to escape analysis [11] for the purpose of detecting race-condition by ensuring objects annotated/mark as thread-local are indeed thread-local (i.e., through the entire execution, they are only reachable from a particular thread t). A similar sort of analysis presented by Boyapati and Rinard [4] proposes extensions to Concurrent Java [10] and uses the auxiliary information to deduce escape information.

There is a wide body of literature on slicing sequential programs, beginning with Weiser’s original paper on slicing [31]. Horwitz et.al[15] proposed a interprocedural program slicing algorithm which has been extended by others to handle various features such as exceptions and unconditional jumps. There has been effort ([20,14,22,21]) in addressing issues involved in slicing program written in object oriented languages with features such as pointers and/or references.

In the realm of slicing concurrent programs, Ramalingam[26] proved the calculation of context-sensitive synchronization-sensitive program slices is undecidable. Later on, Müller-Olm[23] provided tighter lower bounds for slicing synchronization-free programs in the absence of procedures and loops.

Krinke[18] considered intra-procedural slicing for a simple **while**-language with **co-begin/co-end** statements and proposed a form of symbolic execution to prune interference dependences starting from the observation that considering interference dependences to be transitive is overly conservative. Nanda[24] proposed algorithms that he described as context sensitive for slicing concurrent programs. Recently, Krinke[19] proposed a context-sensitive algorithm for slicing concurrent programs. However, these algorithms rely on symbolic execution and/or *may-happen-in-parallel(MHP)* [1,25] information to prune interference dependences. However, in the absence of synchronization operations, the use of MHP algorithms will not provide large reductions in the dependences as it is harder to detect instruction execution ordering. Also, these techniques does not address pruning of interference dependences arising from language features such as dynamically created objects and threads. As the justification for pruning interference dependences are independent (thread locality of objects vs instruction execution ordering), orthogonal pruning techniques such as ours and those mentioned above can be combined to obtain further reductions.

7 Conclusion

We have demonstrated how an existing escape analysis can be adapted and extended to produce information that is useful for reducing the number of edges in dependence graphs used in slicing concurrent Java programs. In addition, the simple addition of what we called entity information to the analysis yields dramatic improvement in precision with very little cost. We believe that lever-

aging escape information in this way is absolutely necessary for scaling slicing of concurrent object-oriented programs to larger code bases.

References

1. J. Aldrich, C. Chambers, E. G. Sirer, and S. J. Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *Proceedings of Static Analysis Symposium (SAS'99)*, pages 19–38, 1999.
2. B. Blanchet. Escape analysis for object-oriented languages: application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999.
3. J. Bogda and U. Hözle. Removing unnecessary synchronization in Java. *ACM SIGPLAN Notices*, 34(10):35–46, 1999.
4. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of 16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, Tampa Bay, FL, USA, Oct 2001.
5. Z. Chen and X. Baowen. Slicing concurrent java programs. *SIGPLAN Notices*, 36(4):41–47, April 2001.
6. J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for object oriented languages. application to Java. In *Proceedings of Conference on Object-Oriented Systems, Languages and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 1–19, Denver, CO, USA, Oct 1999. ACM.
7. J. D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, June 2002.
8. J. D. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical report, IBM Research Division, Thomas J. Watson Research Centre, 2001.
9. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448, June 2000.
10. C. Flanagan and S. N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.
11. C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Proceedings ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, 2001.
12. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, second edition, 2000.
13. J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Lecture Notes in Computer Science*, Sept 1999. Proceedings on the 1999 International Symposium on Static Analysis (SAS'99).
14. S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI'89)*, pages 28–40. ACM, 1989.

15. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI'88)*, volume 23, pages 35–46, 1988.
16. Indus, a toolkit to customize and adapt java programs. This software is available at <http://indus.projects.cis.ksu.edu>.
17. Java grande forum benchmark suite - thread version 1.0. This software is available at http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/. Java Grande Benchmarking Project at Edinburgh Parallel Computing Centre.
18. J. Krinke. Static slicing of threaded programs. In *Proceedings ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM SIGPLAN Notices 33(7).
19. J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of ESEC/SIGSOFT FSE'03*, 2003.
20. L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of International Conference on Software Engineering (ICSE'96)*, pages 495–505, 1996.
21. D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *Proceedings of International Conference on Software Maintenance (ICSM'98)*, pages 358–367, 1998.
22. P. Livadas and A. Rosenstein. Slicing in the presence of pointer variables, 1994.
23. M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing (STOC'01)*, pages 647 – 656, Hersonissos, Greece, 2001. ACM, ACM Press.
24. M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA '00)*, pages 180–190, 2000.
25. G. Naumovich, G. Avrunin, and L. Clarke. An efficient algorithm for computing MHP information for concurrent java programs. Technical Report UM-CS-1998-044, University of Massachusetts, Amherst, October 1998.
26. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):416 – 430, March 2000.
27. V. P. Ranganath. Object-flow analysis for optimizing finite-state models of java software. Master's thesis, Kansas State University, 2002.
28. E. Ruf. Effective synchronization removal for java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 203–213, June 2000.
29. A. V.Aho, R. Sethi, and J. D.Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley International, 1986.
30. R. Vallée-Rai. Soot: A Java Bytecode Optimization Framework. Master's thesis, School of Computer Science, McGill University, Montreal, Canada., Oct 2000.
31. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
32. J. Zhao. Slicing concurrent Java programs. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension (IWPC'99)*, pages 126–133, May 1999.

Data Dependence Profiling for Speculative Optimizations

Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew

Department of Computer Science, University of Minnesota
{tchen, jlin, dai, hsu, yew}@cs.umn.edu

Abstract. Data dependence analysis is the foundation to many reordering related compiler optimizations and loop parallelization. Traditional data dependence analysis algorithms are developed primarily for Fortran-like subscripted array variables. They are not very effective for pointer-based references in C or C++. With more advanced hardware support for speculative execution, such as the advanced load instructions in Intel’s IA64 architecture, some data dependences with low probability can be speculatively ignored. However, such speculative optimizations must be carefully applied to avoid excessive cost associated with potential mis-speculations. Data dependence profiling is one way to provide probabilistic information on data dependences to guide such speculative optimizations and speculative thread generation. Software-based data dependence profiling requires detailed tracing of memory accesses, therefore, could be very time consuming. In this paper, we examine issues related to data dependence profiling, and propose various techniques to improve the efficiency of data dependence profiling. We use the Open Research Compiler (ORC) [15,16] to test the efficiency of our data profiling techniques. We also study the effectiveness of data dependence profiling on data speculative optimizations on Itanium systems. Our results show that efficient data dependence profiling could improve the performance for data speculative optimizations.

1 Introduction

General data dependence analyses [1, 2, 3] have been extensively used for Fortran-like subscripted array expressions in scientific applications. However, those algorithms are mostly inadequate for C and C++ programs due to the widely use of pointer expressions. Existing approaches usually require a very sophisticated inter-procedural pointer alias analysis followed by a shape analysis [4, 5, 6]. The effectiveness of such approaches is often limited because of the increasing use of shared libraries, complex recursive data structures, and dynamically allocated objects in applications [7, 8].

To circumvent such limitations in compiler analyses, some architectural and hardware schemes have been proposed to support data dependence speculation. For example, Intel’s IA64 architecture [9] provides the *advance load* instruction which can be used to schedule a *load* instruction across a possibly aliased *store* instruction. The correctness of the execution is enforced by checking whether such dependences exist or not at runtime. If no such dependence exists, the speculation is successful. If such dependence does exist, the check fails and a respective recovery action will take

place. A similar check and recovery approach has been proposed in speculative multithreaded processors [10, 11].

Like other speculative execution, the cost of mis-speculation is relatively high. To make use of the data speculation, the compiler typically assumes that the mis-speculations should be very rare. However, it is often very difficult, if not possible, for a compiler to obtain such information based only on static analyses algorithms. Simply ignoring all possible data dependences may cause drastic performance degradation in some programs due to frequent mis-speculation and recovery. Like many existing profile-guided optimizations, data dependence profiling could provide the compiler with useful guidance on data speculative optimizations.

In this paper, we present a software-only data dependence profiling approach to study the potential of data dependence profiling. Our data dependence profiling tool is instrumentation based. It is platform independent and can be used for many different types of data speculative execution. Although hardware supported data dependence profiling [12, 13] can be more efficient, they usually are limited by the size of the hardware table, and can only track data dependence among references in a constrained window of execution. Our tool does not have such limitations. It could provide data dependence information for a larger code region, not limited by program structures such as basic blocks, function calls, or loops. The instrumentation-based profiling approach could also provide additional flexibilities. The probability of a data dependence edge, for example, can be defined and, consequently, collected for different optimizations.

Alias profile has been used to guide speculative register promotion in [14]. However, the information provided by the alias profiling is usually less accurate than that by the data dependence profiling. For example, two pointers p and q both point to a memory block allocated from the same *malloc*, but access different parts of the memory block. The alias profiling may indicate that p and q are all pointing to the same memory objects. However, data dependence profiling is based on pair-wise *address* comparison, so that p and q will not be data dependent on each other. We have conducted experiments to show that using data dependence profile yields a higher performance gain than using alias profile on the same speculative optimizations. Furthermore, data dependence profile can be more useful in parallelization than the alias profile. This is because disambiguation among array elements requires finer granularity than a typical alias profile can provide.

Some unique features of our data dependence profiling tool are listed below:

- It detects data dependence quickly. To detect data dependence at runtime, the addresses accessed by each pair of memory references need to be compared. Such cross checking has a complexity of $O(n^2)$, where n is the number of references. Other profiling approaches, such as edge profiling or value profiling, usually have a complexity that is linear to the profiling events. Hence, *efficiency* is very important in data dependence profiling, especially for very large application programs. We use a special hash function in conjunction with some sampling techniques to ensure data dependences can be detected very quickly.
- It handles function calls. Our profiling tool detects data dependences among references within the same function calls, as well as data dependences across function calls. The side effect of function calls can be summarized, and the

compiler can use such information to overcome the barrier of function calls during its optimizations.

- It handles nested loops. For loop related optimizations, it is very important to distinguish loop-carried dependences from loop-independent dependences. In our data dependence profiling, we associate each dependence edge with a distance vector. Since we do not want to limit the collection of data dependence information to the innermost loop only, the profiling tool can generate distance vectors for nested loops simultaneously. Notice that the loops may be nested across procedure boundaries.
- It tracks the dynamic behavior of data dependence in programs. A data dependence edge may occur with different *probabilities* at runtime. However, we found that the *probability* of a data dependence edge needs to be well defined according to the consumers of the profiling information. How to collect the *probability* information in profiling needs to be addressed.

We have implemented our data dependence profiling tool in Intel’s Open Research Compiler (ORC) [15, 16]. The collected data dependence profile can be fed back to the ORC compiler for speculative optimizations. The main contributions of this paper are listed below:

- We present the design and the implementation of an instrumentation-based data dependence profiling tool. This tool is capable of generating detailed data dependence profiles (such as *dependence distance vectors* and *dependence probability*) for nested loops. Different dependence probabilities can be defined and profiled accordingly for different speculative optimizations.
- We discuss and evaluate several techniques to improve the efficiency of data dependence profiling. We use a *shadow* memory space to conduct efficient data dependency checking. Since profiling is merely an approximation of the program behavior, we evaluate various implementation tradeoffs and their impact on the quality and the efficiency of the data dependence profiling. Various sampling techniques to reduce profiling overhead is also studied. Since data dependence is defined among memory references, it needs to consider the program structures, such as procedures and loops. Furthermore, an implementation of sampling technique similar to [17] has reduced the profiling overhead from many times of the original program execution time down to only 20%.
- We show the benefit of data dependence profiling in two optimizations: profile-guided code scheduling and speculative partial redundancy elimination. The performance improvement of some SPEC CPU2000 benchmarks on Itanium machines can be as high as 32%.

The rest of this paper is organized as follows: the benefit of data dependence profiling is shown in section 2; Section 3 describes how the data dependence profiling is performed; The experimental results on how to reduce the overhead of data dependence profiling are presented in section 4; Section 5 discusses the related works; The final section provides the conclusions of this paper.

2 Benefit from Data Dependence Profiling

Before we describe how data dependence profiling is performed, we would like to first show its potential benefit to two common compiler optimizations: the partial redundancy elimination and code scheduling. Our instrumentation tool is built on top of Intel's Open Research Compiler (ORC). The instrumented code is then linked with our data dependence profiling library to generate an executable file. Data dependence is profiled when this executable file is executed with *train* input set. No space reduction or sampling techniques (discussed later) are applied. The profiling results are then fed back to ORC to guide optimizations. The optimized code is executed with the *ref* input set and the execution time is measured. Our baseline is the code compiled with ORC -O3. The reduction in the execution time is reported as performance improvement. The machine used in our experiments is HP workstation i2000 with one 900MHz Itanium2 processor and 2GB of main memory. Our test programs are SPEC CPU2000 benchmarks [18].

2.1 Speculative PRE

The partial redundancy elimination (PRE) [19] includes a set of compiler optimizations such as register promotion, strength reduction and expression redundancy elimination. It is one of the most important optimizations in the compiler. Data speculation has been introduced into PRE using alias profiling information [20, 21]. However, alias profiling often cannot disambiguate among array references, or among memory references in a memory space managed by the application program itself. It also cannot determine whether aliases generated come from the same iteration or from different iterations. Instead, data dependence profiling often can provide more detailed information about memory references than alias profiling.

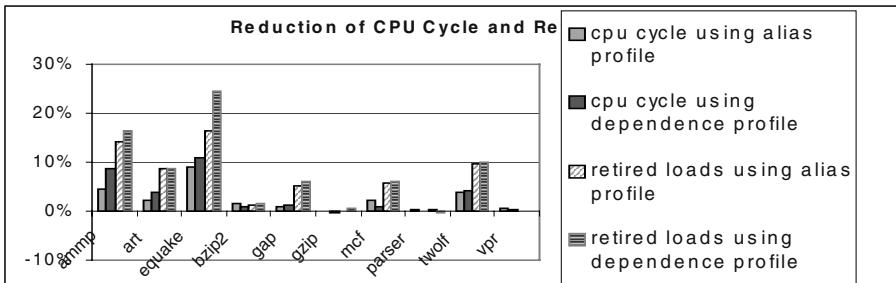


Fig. 1. Improvement on PRE using dependence profiling vs. alias profiling

The speculative PRE using both alias and data dependence profiling information has been implemented on ORC. We measure its performance using *pmon* [22] on Itanium systems. The performance improvement based on alias profiling and dependence profiling is compared in Fig. 1. The total number of CPU cycles and the total number of *loads* retired are measured for each program optimized by the speculative PRE. The result shows that data dependence profiling is able to discover

more redundant loads (more retired loads reduced) and outperforms alias profiling in the total CPU cycle.

2.2 Speculative Code Scheduling

Code scheduling aims to exploit ILP. Data speculation has been used to move *loads* speculatively so that the length of a critical path can be reduced [23]. In the ORC compiler, the speculation is based on heuristic rules. For example, a *load* instruction can be moved across only two *aliased store* instructions (based on the alias analysis in the compiler). We try to conduct more aggressive data speculation based on the data dependence profiling information. Dependence edges with a probability lower than 2% are ignored. This threshold is determined by the overhead of a data mis-speculation on Itanium2: it takes about 50 cycles to jump to and return from its recovery code at a check failure.

Table 1. Improvement on code scheduling using dependence profiling

	Performance improvement	Regular loads changed to speculative loads	Failure rate
quake	8.36%	18.80%	0.34%
art	32.34%	31.32%	0.88%
Mesa	12.23%	9.14%	0.09%
Bzip	0.34%	6.81%	6.98%
Gzip	0.00%	2.45%	0.11%
parser	2.36%	2.92%	2.00%
Vortex	1.94%	4.62%	0.13%
Average	8.22%	10.8%	1.5%

The first column of Table 1 shows the performance improvement on Itanium2 using data dependence profiling information. Some floating-point benchmarks such as *art* can improve performance by 32%. On average, 11% of regular *load* operations are turned into *speculative loads* (reported in the second column) with a very low check failure rate (i.e. less than 1% of mis-speculation) for most benchmark programs, except *bzip* and *parser*. The major reason for the improvement is that, ORC, in fear of the high penalty of mis-speculation, is overly conservative on scheduling data speculative loads. With dependence profiling, more aggressive scheduling can be performed with confidence. The failure rate observed here also shows that the dependence profiling is quite insensitive to different input sets. When we ignore the dependence with less than 2% probability observed with *train* input, only *bzip2* has a failure rate higher than 2% for the execution with *ref* input.

3 Instrumentation-Based Data Dependence Profiling

In this section, we address the issues related to data dependence profiling that include detection of data dependences, handling of function calls and nested loops, and the probability of a data dependence edge.

3.1 Detection of Data Dependences Using Shadow Variables

In our data dependence profiling, we focus only on the data dependence among *memory references*. We ignore data dependence among *registers* because it can be easily obtained by simple static analysis. Therefore, only memory references are instrumented. The instrumented profiling instructions collect the *address value* and the *reference ID* of each instance of memory references at runtime. The reference ID is used to identify each static reference. It is assigned during the instrumentation so that profiling results could be mapped back to the compiler.

Data dependence occurs when two memory references access the *same memory location* in the data space. One occurrence of data dependence can be represented as a *dependence edge* from the *source* to the *sink*. There are four data dependence types: *flow dependence* (or true dependence), *anti-dependence*, *output dependence*, and *input dependence* (or reference dependence).

It is very expensive to check data dependence based on pair-wise address comparison, especially when the program has a large number of memory references. The number of dynamic instances of all memory references could be very large when there are nested loops in the program. Therefore, we use a special data structure, called *shadow memory*, to efficiently detect data dependences. This is a typical trade off between *time* and *space*.

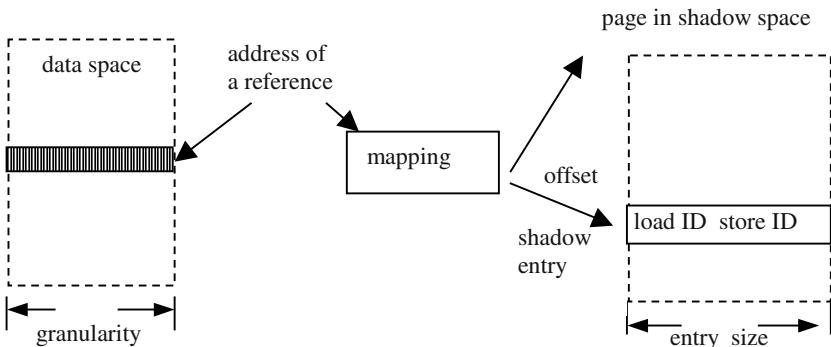


Fig. 2. Detect data dependence using shadow

Shadow is used to store information during profiling. A simple hash function using the *address value* maps each memory reference in the data space to its corresponding shadow entry in the shadow memory (illustrated in Fig. 2). The shadow memory is allocated on demand, and can be freed when the profiling for a particular region such as a particular nested loop or a procedure of interests has completed.

Now, we describe how to detect data dependence during the profiling process. First, each memory reference locates its shadow entry using the hash function on the address value. The shadow entry contains the reference ID of the *latest load* and the *latest store* operation to this memory location, or is *empty* when this location is accessed for the first time. If the memory reference is a *load* operation, there should be a *flow dependence edge* from the latest store to this load operation, and also an *input dependence edge* from the latest load to this load operation. If the memory reference is a *store* operation, there should be an *anti dependence edge* from the latest

load to this store operation, and also an *output* dependence edge from the latest store to this store operation. Finally, the reference ID of the current memory reference is stored into the shadow entry, and the reference ID for previous load or store in the shadow entry is overwritten.

Using this scheme, data dependences can be quickly detected without expensive pair-wise address comparisons among all memory references. Using shadow space is equivalent to a software implementation of the associative memory in [21]. We reduce the number of pair-wise comparisons by focusing only on the execution path of a program, i.e. only the latest load and the latest store are compared. We can use a linked list, for each type of data dependence edges, to record the dependence edges.

3.2 Function Calls

When function calls are present in a profiled program, a dependence edge detected between two memory references in two different procedures needs to be mapped into the common procedure that contains both procedures. It is because, to be useful to a compiler, a data dependence graph of a procedure should contain only the dependences among the memory references and function calls, within the same invocation of the procedure. However, profiling information at runtime may contain a lot of dependences between different invocations of procedures at different call sites. We need to sort through these data dependences and extract relevant data dependence information. It requires the calling path information of all memory references to avoid creating *false* dependence edges.

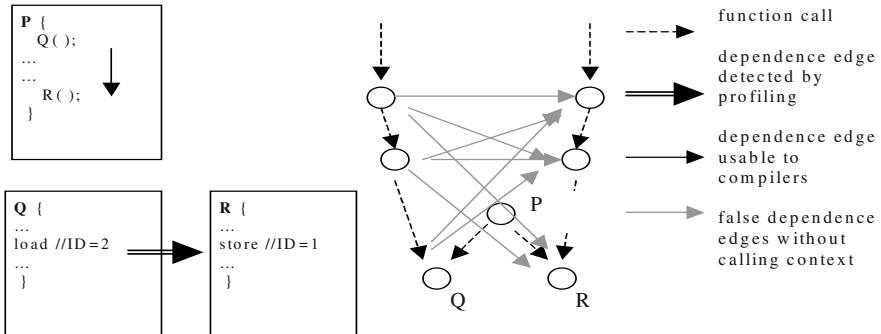


Fig. 3. False dependences may be introduced by function calls

In Fig. 3, for example, a flow dependence from reference 2 to reference 1 occurs only when procedure Q and R are called within procedure P. Therefore, only one dependence edge from the corresponding call site of Q to the call site of R in the procedure P should be generated. If we do not have calling path information for each memory reference, we have to assume that there is a dependence edge from any call site that may calls Q to any call site that may calls R. Here, we include all the procedures calling Q or R directly or indirectly. They are shown as *false* dependence edges in Fig. 3. The example illustrates the need for calling path information in data dependence profiling.

To reduce the number of false dependence edges caused by function calls, the calling context for each memory reference is recorded in its shadow entry. When a dependence edge is detected, their calling contexts help to locate the common procedure. Different kinds of calling context can be used in the dependence profiling, resulting in different precision and efficiency.

- Fully extended call paths. Whenever a procedure is called, a new call node is assigned for this invocation. The call path made up with such call nodes is able to distinguish both the different call sites of a procedure and the different invocations of a call site. It is easy to maintain the fully extended calling path at runtime: push a new node to the call stack when a procedure is invoked, and pop the top node from the call stack when a procedure call returns. The nodes in the call stack can be linked backwardly and we only keep the top node in the shadow entry. When a dependence edge occurs, the common part in their calling context can be identified. A dependence edge should be added in the farthest common procedure from the main procedure between the corresponding references or call sites.
- Compacted call paths. The space requirement of the fully expanded call paths could be very high because a call site may appear in a nested loop or in a recursive call. We could reduce the size of a call path by using only one node to represent multiple invocations of a call site in a nested loop or in recursive calls to reduce the space requirement. Compression techniques [24] can also be used. However, compacted call paths may result in some false dependence edges.

For SPEC CPU2000 benchmarks, even with the train input set, it is still too costly to fully extend all call paths. We use the compact call paths for efficiency.

3.3 Loops

When a dependence edge occurs in a loop nest, a *dependence distance* tells how far the sink is away from the source of a dependence edge in terms of the number of iterations. Optimizations may treat dependences with different distances quite differently. It is important for a profiling tool to be able to generate distance vectors.

In order to generate a distance vector for a dependence edge, we instrument the beginning of the loop body. Each loop is also assigned an iteration counter. This counter is incremented when the beginning of the loop body is encountered. The values in the iteration counters of the loops nested outside of the current reference form an iteration vector. We store the iteration vector in the shadow. When a dependence edge is detected, we first have to find out how the two references nested in loops, with the consideration of the calling context discussed in the previous section. The loop nesting information is not recorded in the shadow. The calling context information in the shadow is used to help identifying the commonly nested loops of the two references. The iteration vectors of the two references are aligned and the iteration counters of their commonly nested loops are identified. The distance vector is computed by subtracting the iteration vector in the shadow from the current iteration vector. Dependences together with their distance vectors are recorded. Depending on the optimizations using the profile, limited distance or the sign of the distance, =, < or >, can be recorded instead for better efficiency.

3.4 Dependence Probability

A static dependence analysis in a compiler usually can only tell whether an edge between two memory references exists or not. However, with the dependence profiling, we can further observe whether the data dependence *rarely*, *frequently*, or *always* occur between the two memory references. Such additional information allows a compiler optimization to deal with a data dependence edge accordingly for a better performance. In general, a compiler can speculate only on those dependences that *rarely* occur in order to avoid costly mis-speculations. Hence, such information is especially important to speculative optimizations in a compiler.

We use a *dependence probability* to depict such dynamic behavior of the dependence. There are several ways to define the *dependence probability*. It depends on how such information is to be used by later speculative optimizations. Here, we only focus on two possible definitions: the *reference-based probability* and the *iteration-based probability*.

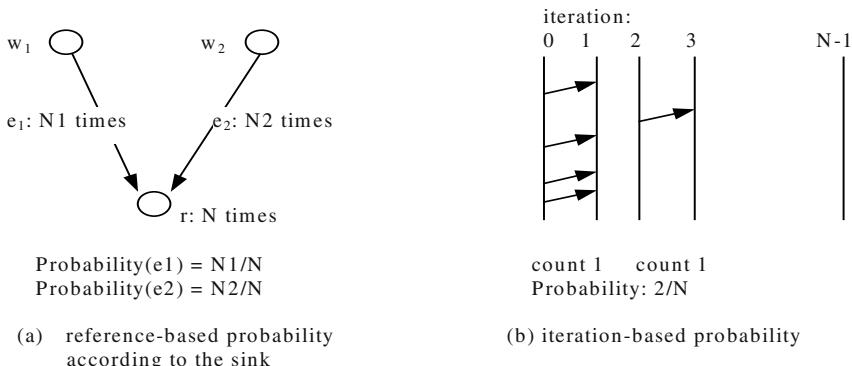


Fig. 4. Different definitions of dependence probability for dependence profiling

The *reference-based probability* is defined as the number of the occurrences of a *dependence edge* over the total number of the occurrence of the *reference*. The *reference* may be either the *source* or the *sink* of the *dependence edge*. This information gives an indication of how often a dependence edge occurs when either the source or the sink of the dependence is referenced during the program execution. In Figure 4(a), an example of the reference-based probability for the sink reference r is illustrated. Assume e_1 and e_2 are two flow dependence edges to the sink reference r . The reference r is executed N times while the edges e_1 and e_2 occur N_1 and N_2 times respectively. The probability of the value coming from reference w_1 is defined as N_1/N and from reference w_2 N_2/N . Such probability is able to tell us which definition is more likely to provide the value to the reference r .

The *iteration-based probability* is defined as the *number of iterations* in which the dependence edge occurs over the total number of iterations of the loop nest, as shown in Figure 4(b). Again, the *iteration-based probability* may be *sink-based* or *source-based*. This information is often related to the dependence distance and the control flow within the loop.

4 How to Reduce the Profiling Overhead

From the discussion in section 3, the data dependence profiling could be quite time consuming because we need to handle each memory reference individually, and there is a significant amount of work on each memory reference. In this section, we present ways to reduce such overhead during the dependence profiling.

The most direct way to reduce such overhead is to reduce the number of memory references that need to be instrumented and tracked. In a typical optimizing compiler such as ORC, many of its optimizations already work toward reducing the number of memory references in the application program. Therefore, our instrumentation is able to make use of the existing optimizations.

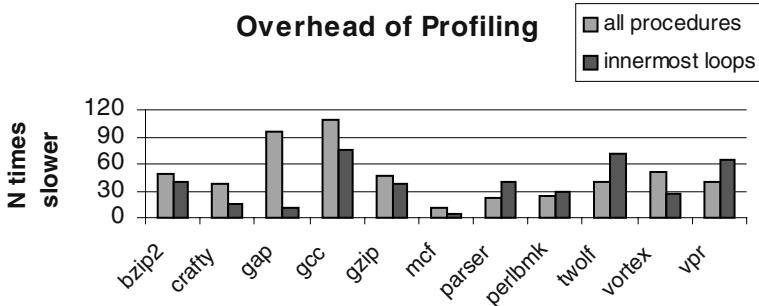


Fig. 5. Overhead of dependence profiling

The data dependence profiling is still quite expensive even when the instrumentation is done after WOPT, in which ORC eliminates most of the redundant memory references. The overhead of various dependence profiling is shown Fig. 5. The average overhead is over 40 times of the original execution time even only for the innermost loops. We need to somehow improve the space and time efficiency to make the tool more usable.

Unfortunately, the tradeoff to reduce the overhead of profiling often involves in losing some *precision* in data dependence profiling. In the following experiments, we will measure both the *efficiency* and the *precision* of each scheme proposed. The efficiency can be measured by the *space requirement* and/or the *profiling time*. We also need to address the precision issues in more details. First, we can measure how many false dependence edges are produced and/or how many true dependence edges are missing after a more efficient profiling method is used. The percentage of such dependence edges with respect to the total dependence edges can be used for such a measure. Secondly, the *dependence probability* of such false/missing dependence edges should also be considered. A dependence edge with lower probability will have a less impact on optimizations because the compiler may speculate it as non-existence.

4.1 Reduce Shadow Size

As described in section 3.1, we trade space for time using shadow during dependence detection. It is important to reduce the shadow size for better space efficiency. We

define the hash function for the shadow in such a way that the conflict in hashing rarely occurs. The total shadow size is determined by: $(\text{size of program space in byte}) * (\text{size of a shadow entry}) / \text{granularity}$. The *granularity* is determined by how many adjacent data bytes would share *one* shadow entry, while the *shadow entry size* is determined by how many bytes are needed to keep the runtime information during the profiling (see Fig. 2). While the program space cannot be changed, the profiling tool can manipulate the *granularity* and the *shadow entry size*.

Since most of the memory references access 4 bytes or 8 bytes each time, the size of total shadow space can be reduced if we let 2^k adjacent data bytes share a shadow entry. However, larger granularities may introduce more false dependences.

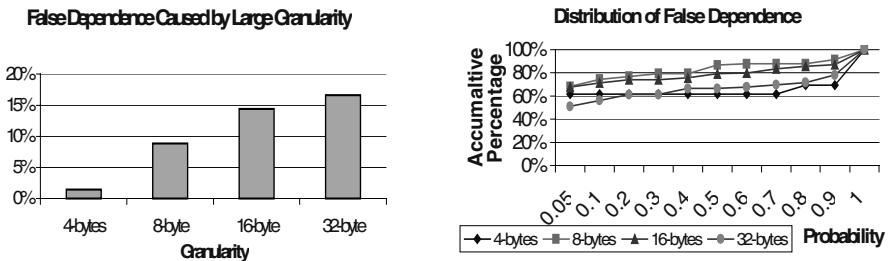


Fig. 6. Precision of different granularities

To study the impact of the granularity, we compare the profiling results with a granularity of 4, 8, 32 and 64 bytes to the results with a granularity of 1 byte in Fig. 6. The results show that the granularity of 4 bytes is quite satisfactory. The reason is that 4-byte data, such as an integer or a 32-bit floating-point number, are the most commonly used. The precision of using a granularity of 16 or 32 bytes is still acceptable. This means that using a cache line for profiling or for communication is still a good compromise.

Another factor affecting the size of shadow is the size of each entry in shadow. From the previous discussion, we know that the reference ID, the calling path and the iteration vector of each reference are stored in each shadow entry. Since compression techniques have been applied to the reference ID and calling path, we focus on the iteration vector here. To uniquely represent iterations of the loops in SPEC2000 benchmarks, the iteration counter in an iteration vector has to be 8 bytes. Hence, a deeply nested loop may need a huge shadow entry. Since only dependences with a short distance, usually a distance of 0 and 1, are important to optimizations, we can store only a partial value of an iteration counter in the shadow. As a result, some false dependence edges may be generated due to the overflow of iteration counters. Fig. 7 reports the precision of iteration counters with a size of 1, 2, and 4 bytes. In this experiment, only the important loops (i.e. the most time-consuming loops) are selected, and only the flow dependence edges with a distance of 0 or 1 are profiled. Though the 1-byte iteration counter may result in a very high percentage of false dependences, most of these false dependences have a probability close to 0, and, hence, could be speculated with little impact on performance. This result helps us to choose 1 byte as the size of our iteration counters.

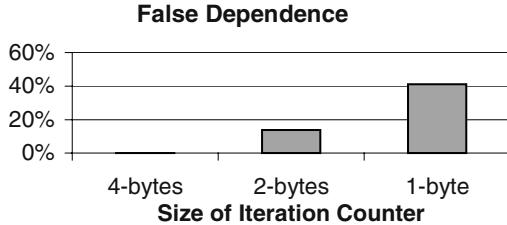


Fig. 7. Precision of different size of iteration vector

4.2 Use Sampling to Improve Time Efficiency

Time efficiency of data dependence profiling could be improved by sampling techniques. In sampling, only a small portion of the events is selected to reduce profiling overhead. The sampling technique has been widely used in many other profiling applications, such as edge profiling, execution time profiling and alias profiling. In these profiling, sampling is directly applied to the relevant events. For example, in edge profiling [25], the direction of each branch (i.e. branch *taken* or *not taken*) is sampled. However, such sampling is not suitable for data dependence profiling. One reason is that the data dependence is a *relation* between two memory references, i.e. its *source* and *sink*. We cannot sample them *independently*. Second reason is that the number of occurrences of a data dependence edge is not directly related to its significance to a certain compiler optimization. Randomly sampling *individual* references may not work for data dependence profiling from a statistical point of view. More organized sampling such as considering a program segment as a whole for sampling is more appropriate. A program segment can be a procedure or a loop.

Most important procedures or loops in our benchmarks are executed many times because they are usually contained in other loop structures and are the most time consuming parts of the program. We can thus use the framework proposed in [17] for more efficient sampling. In this framework, we keep two versions of the program segment to be sampled: one is *with* and the other is *without* instrumentation for dependence profiling. A switch point is set at the entry and the exit of the program segment. We will then switch between these two versions during the execution of the program with a pre-determined sampling rate. Since the profiling is done on the *whole* program segment, instead of *individual* memory references, a more complete snapshot of the data dependences in the program segment can be obtained.

We tested such a sampling technique for both procedures and loops. Different sampling rates are used. The profiling overhead with different sampling rates is measured against the original execution time. The precision of the profiling is measured using the number of missing dependence edges. Such missing dependence edges may in turn introduce some false dependence edges due to the transitive nature of the data dependences. These false edges will not affect the program optimizations. Fig. 8 reports the overhead and the precision of different sample rates for procedures. They are usually in the range from 16% to 167% of the original execution time using

a sampling rate of 0.0001 to 0.1, respectively, and with a precision ranging from 30% to 10% in missing dependence edges.

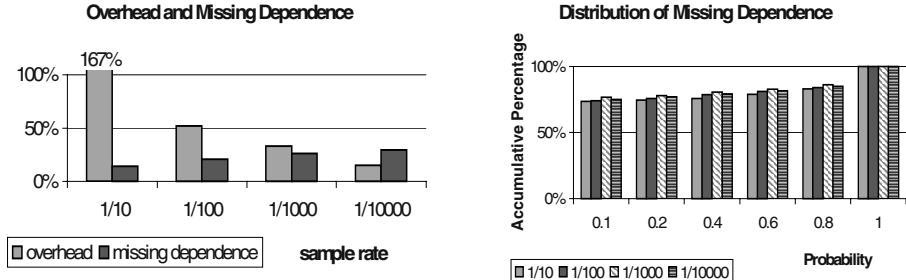


Fig. 8. Sampling of the dependence in procedures

A loop in a procedure may also be invoked many times, and each invocation may have only a few or a lot of iterations. We can take sampling according to the loop invocations. Fig. 9 shows the precision and the efficiency of such sampling techniques. The result shows that the data dependence profiling for loops can have a very high precision even at the rate of 1/256. The overhead can be reduced to below 15% of the original execution time. We can also sample the loop based on loop iterations, i.e. switch between two program versions after a predetermined number of loop iterations. This will incur more overhead than that in invocation-based techniques because of more frequent switching between two versions. However, this technique may be more appropriate for outer loops that are invoked less frequently than the inner loops.

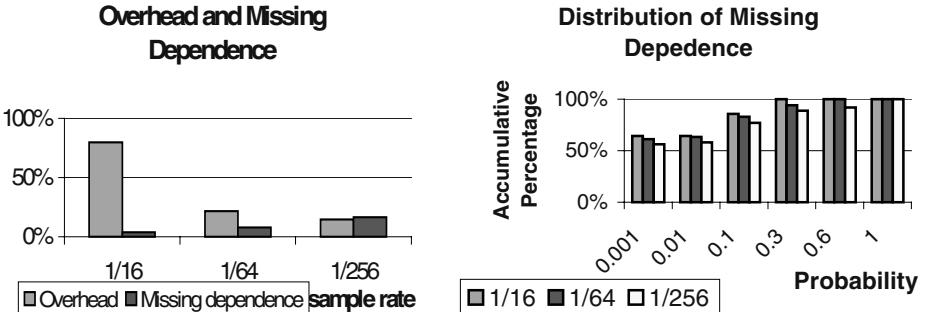


Fig. 9. Loop-oriented sampling

5 Related Work

Dependence profiling is used in [23] to help the optimizations for hyper-blocks. The profiling is supported by a special hardware structure called the *conflict buffer*. The

scheme is quite efficient, compared with our instrumentation-based approach. However, loop structures are not considered in their work. The detailed and useful information about dependences, such as distance vectors and probability, is not generated. The static analysis to gain probability information has been tried in [26], but is limited to array references with linear subscript expressions only. Recent work done by [27] also uses hardware supported profiling to help thread generation. The profiling result is again limited, compared to our approach. Because of the cost of dependence profiling, a co-processor is used to improve the efficiency in [13]. Our instrumentation-based approach, though slower than hardware approach, is a general approach. It can be used when hardware support is not available, and it is more flexible. It is also easier to integrate the compile-time information with the runtime information. With the sampling techniques described in section 4, the overhead of our approach can be reasonably low.

Alias profiling is another type of profiling for memory disambiguation [21]. The dependence profiling is able to provide more accurate information than the alias profiling because the alias profiling is limited by its naming scheme. However, using the finite naming space makes the alias profiling more efficient. The dependence profile can be directly fed into the dependence graph in the compiler. Both dependence profile and alias profile can be fed into the SSA form in the compiler. However, alias profile matches the location factor scheme while the dependence profile is close to statement factor representation.

6 Conclusions

We present a tool for data dependence profiling. This tool is able to produce detailed data dependence information for nested loops and summarized dependence information for function calls. Data dependence profiles can be fed back to compiler to support speculative optimizations.

We observe that the data dependence profiling could be quite helpful in addition to static compiler analysis. Our tool identified a large amount of dependence edges in SPEC CPU2000 benchmarks with a low probability. This information is difficult to obtain using only static compiler analyses. Such information could be very useful for data speculation in optimizations such as PRE and code scheduling.

We also present several schemes to improve the timing and the spatial efficiency of the profiling tool. We use a shadow space with a simple hashing scheme to facilitate fast address comparison for detecting data dependences. It is very important to select appropriate data granularity and the size of shadow entries to minimize the total size of the shadow space.

With our proposed enhancement, we believe data dependence profiling can be very useful for compilers that support data speculative optimizations.

Acknowledgements. This work is supported in part by the U.S. National Science Foundation under grants EIA-9971666, CCR-0105574 and EIA-0220021 and grants from Intel and Unisys.

References

- [1] U. Banerjee, Dependence Analysis for Supercomputing. Kluwer Academic Publishers, 1988.
- [2] Michael Wolfe. Optimizing Supercompilers for Supercomputers. MIT Press, Cambridge MA, 1989.
- [3] Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis; Communications of the ACM, 35, 8 (1992), 102-114.
- [4] Richard L. Kennell and Rudolf Eigenmann. Automatic Parallelization of C by Means of Language Transcription, Proc. of the 11th Int'l Workshop on Languages and Compilers for Parallel Computing, 1998, pages 157--173. Lecture Notes in Computer Science, 1656, pages 166-180.
- [5] L. J. Hendren, J. Hummel, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation, pages 218--229, June 1994.
- [6] R. Ghiya and L.J. Hendren. Is it a tree, a dag, or a cyclic graph? In Symposium on Principle. of Program. Language, New York, NY, January 1996. ACM
- [7] M. Hind. Pointer analysis: Haven't we solved this problem yet? ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pages 54-61, Snowbird, Utah, June 2001.
- [8] Rakesh Ghiya, Daniel Lavery and David Sehr. On the Importance of Points-To Analysis and Other Memory Disambiguation methods For C programs. In Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation, page 47-58, June 2001.
- [9] C. Dulong. The IA-64 Architecture at Work, IEEE Computer, Vol. 31, No. 7, pages 24-32, July 1998.
- [10] G.S. Sohi, S.E. Breach, T. N. Vijaykumar. Multiscalar Processors. The 22nd Annual International Symposium on Computer Architecture, pp. 414-425, June 1995.
- [11] J.-Y. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.-C. Yew. The Superthreaded Processor Architecture. IEEE Transactions on Computers, Special Issue on Multithreaded Architectures, 48(9), September 1999.
- [12] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wenmei W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," Proc. Sixth Int'l Conf. on ASPLOS, October 1994, pp. 183-193.
- [13] C. Zilles and G. Sohi. A programmable co-processor for profiling. In Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7), January 2001. 12.
- [14] R. Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial Redundancy Elimination in SSA Form. ACM Trans. on Programming Languages and systems, v.21 n.3, pages 627-676, May 1999.
- [15] R. D.-C. Ju, S. Chan, and C. Wu. Open Research Compiler (ORC) for the Itanium Processor Family. Tutorial presented at Micro 34, 2001.
- [16] R. D.-C. Ju, S. Chan, F. Chow, and X. Feng. Open Research Compiler (ORC): Beyond Version 1.0, Tutorial presented at PACT 2002
- [17] Matthew Arnold and Barbara G. Ryder A Framework for Reducing the Cost of Instrumented Code SIGPLAN Conference on Programming Language Design and Implementation, 2001.
- [18] <http://www.specbench.org/osg/cpu2000/>
- [19] R. Kennedy, S. Chan, S.-M. Liu, R. Io, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. ACM Trans. Program. Languages and Systems, May 1999.

- [20] J. Lin, T. Chen, W.C. Hsu, P.C. Yew, Speculative Register Promotion Using Advanced Load Address Table (ALAT), In Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization, pages 125-134, San Francisco, California, March 2003
- [21] Jin Lin, Tong Chen, Wei-Chung Hsu, and Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook. Ngai, Sun Chan, A Compiler Framework for Speculative Analysis and Optimizations, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2003
- [22] pfmon project web site: <http://www.hpl.hp.com/research/linux/perfmon/>
- [23] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *Journal of Parallel and Distributed Computing*, 5:334-348, 1988.
- [24] James Larus. Whole program paths. In *Programming Languages Design and Implementation (PLDI)*, 1999.
- [25] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: The showdown. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.
- [26] R. D.-C. Ju, J. Collard, and K. Oukbir. Probabilistic Memory Disambiguation and its Application to Data Speculation, *Computer Architecture News*, Vol. 27, No.1, March 1999.
- [27] Michael Chen and Kunle Olukotun. TEST: A Tracer for Extracting Speculative Threads. In Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization, pages 301-312, San Francisco, California, March 2003

Elkhound: A Fast, Practical GLR Parser Generator^{*}

Scott McPeak and George C. Necula

University of California, Berkeley
`{smcpeak,necula}@cs.berkeley.edu`

Abstract. The Generalized LR (GLR) parsing algorithm is attractive for use in parsing programming languages because it is asymptotically efficient for typical grammars, and can parse with any context-free grammar, including ambiguous grammars. However, adoption of GLR has been slowed by high constant-factor overheads and the lack of a general, user-defined action interface.

In this paper we present algorithmic and implementation enhancements to GLR to solve these problems. First, we present a hybrid algorithm that chooses between GLR and ordinary LR on a token-by-token basis, thus achieving competitive performance for deterministic input fragments. Second, we describe a design for an action interface and a new worklist algorithm that can guarantee bottom-up execution of actions for acyclic grammars. These ideas are implemented in the Elkhound GLR parser generator.

To demonstrate the effectiveness of these techniques, we describe our experience using Elkhound to write a parser for C++, a language notorious for being difficult to parse. Our C++ parser is small (3500 lines), efficient and maintainable, employing a range of disambiguation strategies.

1 Introduction

The state of the practice in automated parsing has changed little since the introduction of YACC (Yet Another Compiler-Compiler), an LALR(1) parser generator, in 1975 [1]. An LALR(1) parser is deterministic: at every point in the input, it must be able to decide which grammar rule to use, if any, utilizing only one token of lookahead [2]. Not every context-free language has an LALR(1) grammar. Even for those that do, the process of modifying a grammar to conform to LALR(1) is difficult and time-consuming for the programmer, and this transformation often destroys much of its original conceptual structure. Nonterminals cease to correspond to sub-languages, and instead come to represent states in the token by token decomposition of the input. Instead of describing the *language* to be parsed, the grammar describes the *process* used to parse it; it's more like

* This research was supported in part by the National Science Foundation Career Grants No. CCR-9875171, No. CCR-0081588, No. CCR-0085949 and No. CCR-0326577, and ITR Grants No. CCR-0085949 and No. CCR-0081588, and gifts from Microsoft Research.

a hand-crafted parsing program, but crammed into Backus-Naur Form. This is very unfortunate, since the grammar is the most important piece of the parsing specification.

The main alternative to a long battle with shift/reduce conflicts¹ is to abandon automatic parsing technology altogether. However, writing a parser by hand is tedious and expensive, and the resulting artifact is often difficult to modify to incorporate extensions, especially those of interest in a research setting. The Edison Design Group C++ Front End includes such a hand-written parser for C++, and its size and complexity attest both to the difficulty of writing a parser without automation and the skill of EDG’s engineers.

This paper describes improvements and enhancements to the Generalized LR (GLR) parsing algorithm that make its performance competitive with LALR(1) parsers, and its programming interface flexible enough for use in a variety of real-world scenarios. These enhancements are demonstrated in the Elkhound parser generator.

1.1 The State of GLR

The Generalized LR parsing algorithm [3,4,5] extends LR by effectively maintaining multiple parse stacks. Wherever ordinary LR faces a shift/reduce or reduce/reduce conflict, the GLR algorithm splits the stack to pursue all options in parallel. One way to view GLR is as a form of the Earley dynamic programming algorithm [6], optimized for use with mostly deterministic² grammars. It can use any context-free grammar, including those that require unbounded lookahead or are ambiguous. Section 2 explains the GLR algorithm in more detail.

GLR is a much more convenient technology to use than LALR(1). It does not impose constraints on the grammar, which allows rapid prototyping. Further, unlike the Earley algorithm, its performance improves as the grammar approaches being deterministic, so a prototype grammar can be evolved into an efficient parsing grammar incrementally.

However, while GLR is asymptotically as efficient as ordinary LR for deterministic input, even mature GLR implementations such as ASF+SDF [7] are typically a factor of ten or more slower than their LALR counterparts for deterministic grammars. Consequently, users are reluctant to use GLR. The poor performance is due to the overhead of maintaining a data structure more complicated than a simple stack, and traversing that data structure to find reduction opportunities.

Existing GLR parsers build a parse tree (or a parse forest, in the case of ambiguous input) instead of executing user-specified actions at each reduction. They build such a tree because it allows the tool to control sharing and the representation of ambiguity. However, because the user cannot control the representation of the parser’s output, applicability is limited. Most commonly, the

¹ A parse state in which the parser cannot decide whether to apply a grammar rule or consume more input is said to have a “shift/reduce” conflict. If the parser cannot decide *which* grammar rule to apply it has a “reduce/reduce” conflict.

² In this context, a grammar is deterministic if it is LALR(1).

user of a parser would like to build an abstract syntax tree (AST) instead of working directly with a parse tree. Or, the analysis task might be simple enough to do during parsing itself. Traversing a parse tree afterwards to simulate on-line reduction actions would once again incur the kinds of performance problems that inhibit adoption.

1.2 Contributions

First, to improve parsing performance on deterministic fragments of grammars, we present an enhancement that allows the parser to dynamically switch between GLR and ordinary LALR(1) at the token level of granularity. The information needed to soundly decide when LALR is sufficient is easily maintained in most cases. Even though it must operate on the GLR data structures, the LALR parser core is much faster because it can make much stronger assumptions about their shape and about the possible sequences of upcoming parsing actions. With this improvement, parsing performance on deterministic grammars is within 10% of good LALR implementations such as Bison [8].

Second, we present a design for a set of user-specified action handlers that gives the user total control over representation decisions, including subtree sharing. Further, we present a crucial modification to the GLR algorithm that ensures the action handlers are executed in a bottom-up order (for acyclic grammars); the original GLR algorithm cannot make this guarantee.

2 The GLR Parsing Algorithm

Since the reader may be unfamiliar with the GLR parsing algorithm [3,4,5], this section contains a brief explanation of the key points. For a more thorough description see [9], an expanded version of this paper.

As with LR³ parsing [10,2], the GLR algorithm uses a parse stack and a finite control. The finite control dictates what parse action (shift or reduce) to take, based on what the next token is, and the stack summarizes the left context as a sequence of finite control state numbers. But unlike LR, GLR’s parse “stack” is not a stack at all: it is a graph which encodes all of the possible stack configurations that an LR parser could have. Each encoded stack is treated like a separate potential LR parser, and all stacks are processed in parallel, kept synchronized by always shifting a given token together.

The encoding of the GLR graph-structured stack (GSS) is simple. Every node has one or more directed edges to nodes below it in some stack, such that every finite path from a top node to the unique bottom node encodes a potential LR parse stack. Figure 1 shows one possible GSS and its encoded stacks. In the case of an ϵ -grammar [11], there may actually be a cycle in the graph and therefore

³ GLR is built on top of the finite control of an LR parser, such as LR(0), SLR(1), LALR(1) or even full LR(1). Elkhound uses LALR(1). In this paper we use the term LR to refer nonspecifically to any of these.

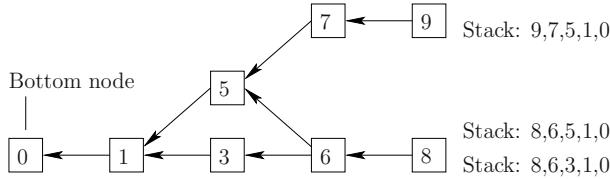


Fig. 1. An example graph-structured stack. Each node is labeled with its parse state number.

an infinite number of paths; Elkhound can handle ϵ -grammars, but they are not further considered in this paper.

The GLR algorithm proceeds as follows: On each token, for each stack top, every enabled LR action is performed. There may be more than one enabled action, corresponding to a shift/reduce or reduce/reduce conflict in ordinary LR. A shift adds a new node at the top of some stack node. A reduce also adds a new node, but depending on the length of the production’s right-hand side, it might point to the top or into the middle of a stack. The latter case corresponds to the situation where LR would pop nodes off the stack; but the GLR algorithm cannot in general pop reduced nodes because it might *also* be possible to shift. If there is more than one path of the required length from the origin node, the algorithm reduces along all such paths. If two stacks shift or reduce into the same state, then the stack tops are merged into one node. In Figure 1, the node with state 6 is such a merged node.

When the algorithm performs a reduction, it executes the user’s action code. The result of an action is called a *semantic value*, and these values are stored on the links between the stack nodes.⁴ When a reduction happens along a particular path, the semantic values stored on the links in that path are passed as arguments to the reduction action. If two different reductions lead to the same configuration of the top two stack nodes, i.e. the resulting stacks use the same final GSS link, the algorithm merges their top-most semantic values. Each of the merged values corresponds to a different way of reducing some sequence of ground terminals to a particular nonterminal (the same nonterminal for both stacks): an ambiguity. The merged value is then stored back into the link between the top two stack nodes, and participates in future reduction actions in the ordinary way.

3 GLR/LR Hybrid

When using a parsing grammar for a programming language, the common case for the GLR algorithm is to have only one top stack node, and one possible parse action. That is, for most of the input, the ordinary LR algorithm would suffice. For our C++ grammar about 70% of the parse actions fit this description.

⁴ It would be incorrect to store values in the stack nodes themselves, because a node at the top of multiple stacks must have a distinct semantic value for each stack.

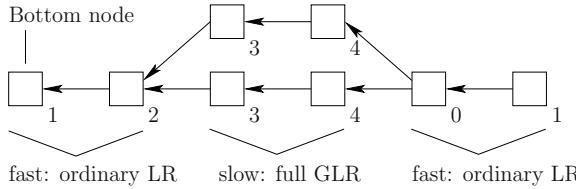


Fig. 2. In this graph-structured stack, each node is labeled with its deterministic depth.

It would therefore be profitable to use LR when possible because performing reductions is much simpler (and therefore faster) with LR, and reductions account for the bulk (about 80% for the C++ grammar) of the actions during parsing. The main cause for slower reductions with GLR is the need to interpret the graph-structured stack: following pointers between nodes, iteration over nodes' successors, and the extra mechanism to properly handle some special cases [5,9] all add significant constant-factor overhead. Secondary causes include testing the node reference counts, and not popping and reusing node storage during reductions.

To exploit LR's faster reductions, we need a way to predict when using LR will be equivalent to using GLR. Clearly, LR can only be used if there is a unique top stack node, and if the action for the current token at the top node's state is unambiguous. If that action is a shift, then we can do a simple LR shift: a new top node is created and the token's semantic value is stored on the new link.

However, if the action is a reduce, then we must check to see if the reduction can be performed more than once (via multiple paths), because if it can, then the GLR algorithm must be used to make sure that all possible reductions are considered. To enable this check, we modified the algorithm to keep track of each node's *deterministic depth*, defined to be the number of stack links that can be traversed before reaching a node with out-degree greater than one. The bottom node's depth is defined to be one. Figure 2 shows an example. Any time the enabled reduction's right-hand side length (call it n) is less than or equal to the top node's deterministic depth, the reduction will only touch parts of the stack that are linear (in-degree and out-degree at most one). Therefore a simple reduction can be performed: deallocate the top n nodes, and create in their place one new node whose link will hold the reduction's semantic value.

Maintaining the deterministic depth is usually easy. When a node is created its depth is set to one more than that of its successor. When a second outgoing link is added to a node, its depth is reset to zero. When a node's depth is reset, if any nodes point at it, their depths are also potentially affected and so must be recomputed. This happens rarely, about 1 in every 10,000 parse actions for the C++ grammar, and can be done efficiently with a topological sort (details omitted due to space constraints).

An important property of the hybrid scheme is that it ensures that the parsing performance of a given sub-language is independent of the context in which

it is used. If we instead tried the simpler approach of using LR only when the stack is *entirely* linear, then (unreduced) ambiguity anywhere in the left context would slow down the parser. For example, suppose a C++ grammar contains two rules for function definitions, say, one for constructors and another for ordinary functions. If these rules have an ambiguity near the function name, that ambiguity will still be on the stack when the function body is parsed. By allowing ordinary LR to be used for the body despite the latent ambiguity, the parsing performance of the statement language is the same in any context. As a result, the effort spent removing conflicts from one sub-language is immediately beneficial, without having to chase down conceptually unrelated conflicts elsewhere.

As shown in Section 6.1, the hybrid algorithm is about five times faster than the plain GLR algorithm for grammars that are LALR(1) or inputs that exercise only the LALR(1) fragment of a grammar.

4 User-Specified Actions

The GLR algorithm’s flexibility provides two basic challenges to any implementation that associates arbitrary user code with the reduction actions. First, while alternative parses are being pursued, semantic values are *shared* between the alternatives. Second, if multiple parse trees can be constructed for a region of the input, the semantic values from the different interpretations have to be *merged*.

As a running example, we use an ambiguous grammar for sums:

$$E \rightarrow E + E \mid b \quad (EEb)$$

Figure 3 shows a parse forest for the input “ $b + b + b'$ ”.

4.1 Sharing Subtrees

When a reduction action yields (produces as its return value for use by subsequent reductions) a semantic value such as v_1 , the value is stored in a link between two stack nodes. If that link is used for more than one subsequent reduction, then v_1 will be passed as an argument to more than one action; in Figure 3, v_1 has been used in the creation of v_4 and v_6 .

Depending on what the actions do, there may be consequences to sharing that require user intervention. For example, if explicit memory management is being used, unrestricted sharing could lead to objects being deallocated multiple times; or, if the analysis is trying to count instances of some construct, sharing could cause some subtrees to be counted more than once.

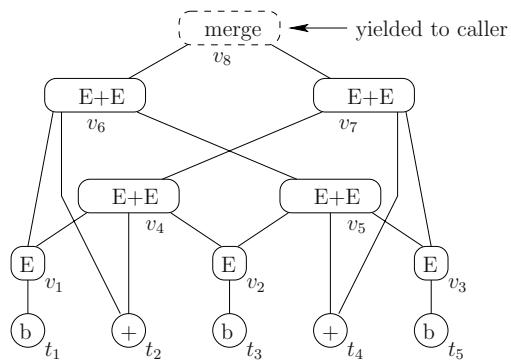


Fig. 3. A parse forest for the EEb grammar.

To allow for a range of sharing management strategies, Elkhound allows the user to associate with each symbol (terminal and nonterminal) two functions, `dup()` and `del()`. `dup(v)` is called whenever v is passed to a reduction action, and its return value is stored back into the stack node link for use by the next action. In essence, the algorithm surrenders v to the user, and the user tells the algorithm what value to provide next time. When a node link containing value v is deallocated, `del(v)` is called. This happens when the last parser that could have potentially used v 's link fails to make progress. Note that these rules also apply to semantic values associated with terminals, so t_2 and t_4 will be properly shared. As a special case, the calls to `dup` and `del` are omitted by the ordinary LR core, since semantic values are always yielded exactly once (`dup` immediately followed by `del`) in that case.

Typical memory management strategies are easy to implement with this interface. For a garbage collector, `dup()` is the identity function and `del()` does nothing (this is the default behavior in Elkhound). To use reference counting, `dup()` increments the count and `del()` decrements it. Finally, for a strict ownership model, `dup(v)` makes a deep copy of v and `del()` recursively deallocates. This last strategy is fairly inefficient, so it should probably only be used in a grammar with at most occasional nondeterminism. In any case, the design neither requires nor prohibits any particular memory management strategy.

4.2 Merging Alternatives

If the grammar is ambiguous, then some inputs have more than one parse tree. In that case, semantic values representing the competing alternatives for the differing subtrees must be merged, so each nonterminal has an associated `merge()` function in the Elkhound API. For example, in Figure 3, semantic values v_6 and v_7 arise from different ways of parsing the same sequence of ground terminals, so the algorithm calls `merge(v_6, v_7)` and stores the return value v_8 back into the stack node link for use by future reduction actions.

Now, the user has at least three reasonable options in a `merge(v_6, v_7)` function: (1) pick one of the values to keep and discard the other one, (2) retain the ambiguity by creating some explicit representation of its presence, or (3) report an error due to unexpected input ambiguity. Option (3) is of course easy to do.

Unfortunately, options (1) and (2) don't always work in conventional GLR implementations, because reductions and merges are not always performed in a bottom-up order. As shown below, it is possible to yield a semantic value to a reduction action, only to then discover that the yielded value was ambiguous and needed to be merged. Because of this order violation, actions are severely constrained in what they can do, because they are given incomplete information.

To illustrate the problem, consider the grammar *SAdB* and the GLR algorithm's activities while parsing “*d*”:

$$\begin{array}{l} S \rightarrow A \\ A \rightarrow d \quad | \quad B \\ B \rightarrow d \end{array} \quad (\textit{SAdB})$$

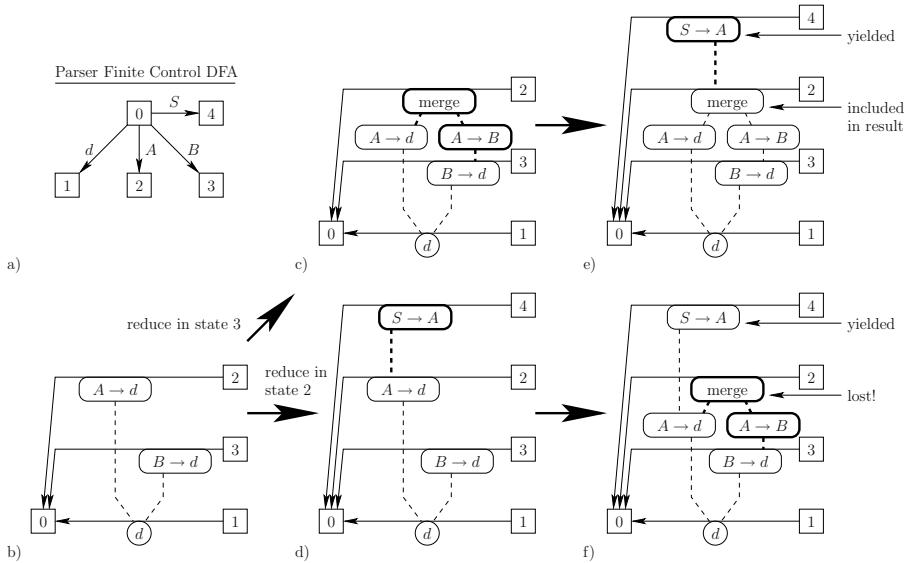


Fig. 4. There are two possible reduction sequences for the *SAdb* grammar, depending on reduction order. Square boxes are stack nodes, and rounded rectangles represent the result of running the user’s reduction action code. Dotted lines show the flow of semantic values; they resemble parse tree edges.

For reference, Figure 4a shows the states of the finite control. In Figure 4b, the *d* has been shifted, creating the node with state 1, and the actions for $A \rightarrow d$ and $B \rightarrow d$ have been executed, creating nodes with states 2 and 3. The dotted lines reflect dependencies between semantic values. But from there the algorithm can proceed in two ways because there are two top nodes that can reduce.

First, Figures 4c and 4e show the sequence of GLR stack configurations when the node in state 3 reduces first. First (Figure 4c), the action for $A \rightarrow B$ runs. Then, because this reduction leads to another stack configuration with state 2 on top of (pointing to) state 0, the algorithm merges the two semantic values for nonterminal *A* as well as the stacks themselves. Finally, (Figure 4e) the action for $S \rightarrow A$ runs. The semantic value corresponding to $A \rightarrow B$ participates in the final result because *A* was merged before it was passed to the action for $S \rightarrow A$. No information has been lost, and this is good.

On the other hand, Figures 4d and 4f show an alternative sequence, where state 2 reduces first. In that case, the action for $S \rightarrow A$ runs immediately (Figure 4d), and this is in fact the final result of the parse. Then (Figure 4f), the action for $A \rightarrow B$ and the `merge()` for *A* run, but nothing more is done with either value; the parser has already performed the reductions corresponding to state 2. The effect of one of the possible parses is lost, which is bad.

The problem stems from the fact that conventional GLR does the parsing work for each token using a worklist of stack nodes. Since each node is processed in its entirety, performing all reductions enabled at that node before the reductions for any other node, certain reductions are forced to occur together. Thus, it isn't always possible to pick an order for processing the stack nodes that will ensure bottom-order action execution.

To guarantee bottom-up actions, we must use a finer-grained worklist. Specifically, we will put *reduction* opportunities themselves in the list, instead of stack nodes. As each stack node is created, the algorithm computes the set of enabled actions. Shifts are postponed until the end of the phase (as in conventional GLR), and reductions are inserted into the worklist. The reduction worklist is maintained in a sorted order, according to this partial order:

- Rule 1. Reductions that span fewer tokens come first.
- Rule 2. If two reductions $A \rightarrow \alpha$ and $B \rightarrow \beta$ span the same tokens, then $A \rightarrow \alpha$ comes first if $B \rightarrow^+ A$.

In Figure 4, Rule 2 would force $A \rightarrow B$ to execute before $S \rightarrow A$ since $S \rightarrow^+ A$. In general, reductions executed in this order are guaranteed to be bottom-up, assuming the grammar is acyclic.⁵ The proof has been omitted, but appears in [9].

It should be noted that traditional GLR implementations cope with the action order problem by the way they construct their parse trees. For example, the algorithm described in [5] uses “Symbol” nodes to represent semantic values, which can be updated after being yielded since they are always incorporated by reference. Since the user's code is not run during parsing, it cannot observe that the semantic value is temporarily incomplete. We think that requiring users of Elkhound to write actions that are similarly tolerant would be too burdensome.

5 Case Study: A C++ Parser

To verify its real-world applicability, we put Elkhound to the test and wrote a C++ parser. This effort took one of the authors about three weeks. The final parser specification is about 3500 non-blank, non-comment lines, including the grammar, abstract syntax description and post-parse disambiguator (a limited type checker), but not including support libraries. The grammar has 37 shift/reduce conflicts, 47 reduce/reduce conflicts and 8 ambiguous nonterminals.

This parser can parse and fully disambiguate most⁶ of the C++ language, including templates. We used our implementation to parse Mozilla, a large (about 2 million lines) open-source web browser.

⁵ If the grammar is cyclic then Rule 2 is not necessarily consistent, since it could be that both $B \rightarrow^+ A$ and $A \rightarrow^+ B$.

⁶ Namespaces ([12] Section 7.3) and template partial specialization ([12] Section 14.5.4) are not currently implemented because they are not needed to parse Mozilla. We foresee no new difficulties implementing these features.

The C++ language definition [12] includes several provisions that make parsing the language with traditional tools difficult. In the following sections we explain how we resolved these parsing difficulties using the mechanisms available in Elkhound.

5.1 Type Names versus Variable Names

The single most difficult task for a C or C++ parser is distinguishing type names (introduced via a `typedef`) from variable names. For example, the syntax “`(a)&(b)`” is the bitwise-and of `a` and `b` if `a` is the name of a variable, or a type-cast of the expression `&b` to type `a` if `a` is the name of a type.

The traditional solution for C, sometimes called the “lexer hack,” is to add type names to the symbol table during parsing, and feed this information back into the lexical analyzer. Then, when the lexer yields a token to the parser, the lexer categorizes the token as either a type name or a variable name.

In C++, the hack is considerably more difficult to implement, since `a` might be a type name *whose first declaration occurs later in the file*: type declarations inside a class body are visible in all method definitions of that class, even those which appear textually before the declaration. For example:

```
int *a;                                // variable name (hidden)
class C {
    int f(int b) { return (a)&(b); }      // cast!
    typedef int a;                        // type name (visible)
};
```

To make lexer hack work for C++, the parser must defer parsing of class method bodies until the entire class declaration has been analyzed, but this entails somehow saving the unparsed method token sequences and restarting the parser to parse them later. Since the rules for name lookup and introduction are quite complex, a great deal of semantic infrastructure is entangled in the lexer feedback loop.

However, with a GLR parser that can tolerate ambiguity, a much simpler and more elegant approach is possible: simply parse every name as *both* a type name and a variable name, and store both interpretations in the AST. During type checking, when the full AST and symbol table are available, one of the interpretations will fail because it has the wrong classification for a name. The type checker simply discards the failing interpretation, and the ambiguity is resolved. The scoping rules for classes are easily handled at this stage, since the (possibly ambiguous) AST is available: make two passes over the class AST, where the first builds the class symbol table, skipping method bodies, and the second pass checks the method bodies.

5.2 Declarations versus Statements

Even when type names are identified, some syntax is ambiguous. For example, if `t` is the name of a type, the syntax “`t(a);`” could be either a declaration of a

variable called `a` of type `t`, or an expression that constructs an instance of type `t` by calling `t`'s constructor and passing `a` as an argument to it. The language definition specifies ([12], Section 6.8) that if some syntax can be a declaration, then it is a declaration.

Using traditional tools that require LALR(1) grammars, and grammars for language fragments A and B , we would need to write a grammar for the language $A \setminus B$. This is at best difficult, and at worst impossible: the context-free languages (let alone LALR(1) languages) are not closed under subtraction.

The solution in this case is again to represent the ambiguity explicitly in the AST, and resolve it during type checking. If a statement can either be a declaration or an expression, then the declaration possibility is checked first. If the declaration is well-formed (including with respect to names of types vs. names of variables) then that is the final interpretation. Otherwise the expression possibility is checked, and is used if it is well-formed. If neither interpretation is well-formed, then the two possible interpretations are reported to the user, along their respective diagnostic messages.

5.3 Angle Brackets

Templates (also known as polymorphic classes, or generics) are allowed to have integer arguments. Template arguments are delimited by the angle brackets `<` and `>`, but these symbols also appear as operators in the expression language:

```
template <int n> class C { /*...*/ };
C< 3+4 > a;                                // ok; same as C<7> a;
C< 3<4 > b;                                // ok; same as C<1> b;
C< 3>4 > c;                                // syntax error
C< (3>4) > d;                                // ok; same as C<0> d;
```

The language definition specifies that there cannot be any unparenthesized greater-than operators in a template argument ([12], Section 14.2, par. 3). By recording grouping parentheses in the AST, we can select the correct parse with a simple pattern check once all possibilities have been parsed.

A correct implementation in an LALR(1) setting would again require recognizing a difference between languages. In this case it would suffice to split the expression language into expressions with unparenthesized greater-than symbols and expressions without them. It is interesting to note that, rather than endure such violence to the grammar, the authors of gcc-2.95.3 chose to use a precedence specification that works most of the time but is wrong in obscure cases: for example, gcc cannot parse the type “`C< 3&&4 >`”. This is the dilemma all too often faced by the LALR(1) developer: sacrifice the grammar, or sacrifice correctness.

5.4 Debugging Ambiguities

Building the C++ parser gave us a chance to explore one of the potential drawbacks of using the GLR algorithm, namely the risk that ambiguities would end

up being more difficult to debug than conflicts. After all, at least conflicts are decidable. Would the lure of quick prototyping lead us into a thorn bush without a decidable boundary?

Fortunately, ambiguities have not been a problem. By reporting conflicts, Elkhound provides hints as to where ambiguities may lie, which is useful during initial grammar development. But as the grammar matures, we find ambiguities by parsing inputs, and understand them by printing the parse forest (an Elkhound option). This process is fundamentally different and more natural than debugging conflicts, because ambiguities and parse trees are concepts directly related to the *grammar*, whereas conflicts conceptually depend on the parsing *algorithm*. Consequently, debugging ambiguities fits easily among, and is in fact a very small portion of, other testing activities.

6 Performance

In this section we compare the performance of Elkhound parsers to those of other parser generators, and also measure in detail the parsing performance of the C++ parser (itself written in C++), to test the claim that our algorithm enhancements in fact yield competitive performance. The experiments were performed on a 1GHz AMD Athlon running Linux. We report the median of five trials.

6.1 Parser Generators

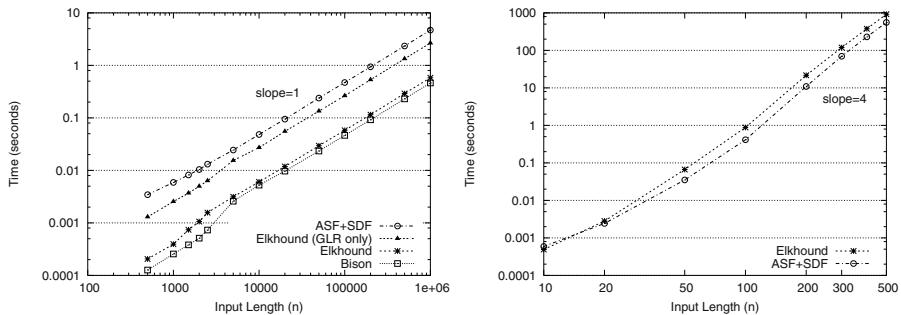
For comparison with an LALR(1) implementation, we compare Elkhound to Bison, version 1.28 [8]. Bison associates user action code with each reduction, and generates a parser written in C.

For comparison with an existing GLR implementation, we used the ASF+SDF Meta-Environment [7]. The Meta-Environment contains a variety of language-processing tools, among which is a scannerless GLR parser [13]. We used the Meta-Environment bundle version 1.1.1, which contains version 3.7 of the SGLR component. This package is written in C.

To measure the speed of Elkhound's ordinary LR parser, we measured its performance against Bison and ASF+SDF on the following LALR(1) grammar, also used in [14]:

$$\begin{array}{c} E \rightarrow E + F \mid F \\ F \rightarrow a \end{array} \quad (EFa)$$

All parsers were run with empty actions, i.e. as recognizers, and using trivial (one character is a token) lexers. As shown in Figure 5a, the Elkhound LR parser core is only about 10% slower than Bison, whereas ASF+SDF is a factor of ten slower than both. When Elkhound's hybrid mechanism (Section 3) is disabled, parsing slows down by a factor of five. This validates the hybrid design: the overhead of choosing between LR and GLR is almost negligible, and the speed improvement when LR is used is substantial.



a) EFa grammar, input is $a(+a)^n$. b) EEb grammar, input is $b(+b)^n$.

Fig. 5. Parser performance (log-log scale).

To measure the full GLR parser, we also measured performance on the highly ambiguous EEb grammar, reproduced here:

$$E \rightarrow E + E \mid b \quad (EEb)$$

This grammar generates the language described by the regular expression $b(+b)^*$.

As shown in Figure 5b, Elkhound's performance is very similar to that of ASF+SDF. This suggests that the algorithm modifications previously presented, especially the use of a worklist of reductions instead of stack nodes (Section 4.2), does not substantially compromise GLR performance. Note that both require $\Theta(n^4)$ time; for grammars with such a high degree of ambiguity, GLR is often slower than the Earley algorithm [6].

6.2 C++ Parser Performance

To test the C++ parser and measure its performance, we used it to parse Mozilla 1.0. Mozilla has about 2000 source modules in its Linux configuration, averaging about 30000 preprocessed lines each. We selected six of Mozilla's modules at random to measure.

Table 1 shows several measurements for each file. Parsing time is reported in milliseconds. “No-LR” is parse time when the LR parser is disabled, and (×) is the ratio of No-LR to Parse. The ratios show that while the GLR/LR hybrid technique is certainly beneficial, saving about a factor of two, it is not as effective for the C++ grammar as it is for a completely deterministic grammar such as EFa , where it saves a factor of five. Of course, the reason is that the C++ parser cannot use the LR parser all the time; on the average, for our current grammar, it can use it only about 70% of the time. Further effort spent removing conflicts would presumably raise this percentage.

“Disamb” is the time for the post-parse disambiguator to run; the sum of the Parse and Disamb columns is the total time to parse and disambiguate. “gcc-

Table 1. C++ Parser Performance (times in ms).

Preprocessed File Name	Lines	Elkhound			gcc-2.95.3	gcc-3.3.2
		Parse	No-LR (×)	Disamb	Parse (×)	Parse (×)
nsUnicodeToTeXCMRt1.i	9537	16	36 (2.25)	50	60 (1.10)	80 (0.83)
nsAtomTable.i	19369	104	179 (1.72)	296	270 (1.48)	480 (0.83)
nsCLiveconnectFactory.i	24055	80	167 (2.09)	273	250 (1.41)	230 (1.53)
nsSOAPPropertyBag.i	26807	173	298 (2.30)	418	460 (1.28)	740 (0.80)
nsMsgServiceProvider.i	39215	209	378 (1.81)	545	560 (1.35)	1270 (0.59)
nsHTMLEditRules.i	49566	495	827 (1.67)	934	1140 (1.25)	2170 (0.66)

2.95.3” and “gcc-3.3.2” are the time for two versions of gcc to parse the code as measured by its internal `parse_time` instrumentation, and (×) is the ratio of Elkhound total parse time to g++ parse time. Remarkably, the Elkhound C++ parser is typically only 30–40% slower than gcc-2, and usually faster than gcc-3.

7 Related Work

Performance. In the worst case, the GLR algorithm takes $O(n^{p+1})$ time, where p is the length of the longest right-hand side. Converting the grammar to Chomsky normal form would thus make the bound $O(n^3)$ but would destroy conceptual structure. Kipps [15] suggests a different way to achieve $O(n^3)$ bounds, but with high constant-factor costs.

User-specified Actions. ASF+SDF intends its users to write analyses using a sophisticated algebraic transformation (rewrite) engine. While the actions are not arbitrary, they are nonetheless powerful. The advantage is the availability of a number of high-level notations and tools; the disadvantage is the impedance mismatch with components outside the ASF+SDF framework. In particular, its internal ATerm [16] library prohibits destructive updates.

Bison allows user-specified actions, and recent versions of Bison include an extended parser that emulates GLR by copying the stack instead of building a graph, and executing user actions by traversing the parse forest afterwards. Copying the stack leads to exponential worst-case time, and tree traversal is expensive. It’s intended for grammars with only minor deviations from LALR(1).

Disambiguation. Many systems support some form of declarative disambiguation. Several static conflict resolution schemes such as precedence and associativity [17] or follow restrictions [18] are implemented in Elkhound. Since it uses a conventional LR finite control, it is straightforward to support such schemes.

Dynamic, parse-time disambiguation in Elkhound is supported through the `keep` and `merge` functions. ASF+SDF has related `reject` and `prefer` directives. `reject` does declaratively what `keep` does imperatively, namely to recognize a language difference such as $A \setminus B$. When B is already described as a nonterminal in the grammar, `reject` is more convenient; when B is ad-hoc or context-

dependent, the flexibility of `keep` is handy. Similarly, `prefer` can be simulated imperatively by `merge`, but `merge` can also choose to retain the ambiguity.

Wagner and Graham [19] argue for post-parse semantic disambiguation, in an incremental GLR parsing setting. We follow their lead with a batch parser.

Other Algorithms. The first parsing algorithm for the entire class of context free grammars was the Earley dynamic programming algorithm [6], with running time $\Omega(n^2)$ and $O(n^3)$. A number of variations and refinements have been proposed [20,21,14,22,23], but none has yet emerged as a practical algorithm for parsing programming languages.

By adding backtracking to a deterministic algorithm, one can achieve the effect of unbounded lookahead. Conventional systems that do this include BtYacc and ANTLR; one can also use higher-order combinators in a functional language to do the same [24]. However, since they do not yield all parses for an ambiguous grammar, many of the techniques presented in Section 5 would not be possible.

8 Conclusion

This paper presents two key enhancements to the GLR algorithm. First, the GLR/LR hybrid substantially improves performance for deterministic grammar fragments. Second, the use of a reduction worklist instead of a node worklist enables user-defined actions to be executed in bottom-up order.

We then demonstrate the benefits of each improvement by the process of constructing a parser for C++. User-written actions, especially during ambiguity merging, are capable of effectively disambiguating troublesome constructs, and the resulting parser comes close to matching the speed of a production compiler.

We believe GLR parsing is a valuable tool for language research, and towards that end Elkhound and its C++ parser have been released under an open-source (BSD) license, available at <http://www.cs.berkeley.edu/~smcpeak/elkhound>.

References

1. Johnson, S.C.: YACC: Yet another compiler compiler. In: UNIX Programmer’s Manual (7th edn). Volume 2B. (1979)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986)
3. Lang, B.: Deterministic techniques for efficient non-deterministic parsers. In Loeckx, J., ed.: Automata, Languages and Programming. Volume 14 of Lecture Notes in Computer Science. Springer (1974) 255–269
4. Tomita, M.: Efficient Parsing for Natural Language. Int. Series in Engineering and Computer Science. Kluwer (1985)
5. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands (1992)
6. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13** (1970) 94–102

7. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. *SIGPLAN Notices* **24** (1989) 43–75
8. Donnelly, C., Stallman, R.M.: Bison: the YACC-compatible Parser Generator, Bison Version 1.28. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139 (1999)
9. McPeak, S.: Elkhound: A fast, efficient GLR parser generator. Technical Report CSD-02-1214, University of California, Berkeley (2002)
10. Knuth, D.E.: On the translation of languages from left to right. *Information and Control* **8** (1965) 607–639
11. Nozohoor-Farshi, R.: GLR parsing for ϵ -grammars. In Tomita, M., ed.: Generalized LR Parsing. Kluwer (1991) 61–75
12. International Organization for Standardization: ISO/IEC 14882:1998: Programming languages — C++. International Organization for Standardization, Geneva, Switzerland (1998)
13. Visser, E.: Scannerless generalized-LR parsing. Technical Report P9707, University of Amsterdam (1997)
14. Alonso, M.A., Cabrero, D., Vilares, M.: Construction of efficient generalized LR parsers. In: WIA: International Workshop on Implementing Automata, LNCS, Springer-Verlag (1997)
15. Kipps, J.R.: GLR parsing in time $O(n^3)$. In Tomita, M., ed.: Generalized LR Parsing. Kluwer (1991) 43–60
16. van den Brand, M., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient annotated terms. *Software Practice and Experience* **30** (2000) 259–291
17. Earley, J.: Ambiguity and precedence in syntax description. *Acta Informatica* **4** (1975) 183–192
18. van den Brand, M., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Compiler Construction. (2002) 143–158
19. Wagner, T.A., Graham, S.L.: Incremental analysis of real programming languages. In: ACM Programming Language Design and Implementation (PLDI). (1997) 31–43
20. Graham, S.L., Harrison, M.A., Ruzzo, W.L.: An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **2** (1980) 415–462
21. McLean, P., Horspool, R.N.: A faster Earley parser. In: Compiler Construction. (1996) 281–293
22. Schröer, F.W.: The ACCENT compiler compiler, introduction and reference. Technical Report 101, German National Research Center for Information Technology (2000)
23. Aycock, J., Horspool, R.N., Janoušek, J., Melichar, B.: Even faster generalized LR parsing. *Acta Informatica* **37** (2001) 633–651
24. Hutton, G.: Higher-order functions for parsing. *Journal of Functional Programming* **2** (1992) 323–343

Generalised Parsing: Some Costs

Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos

Royal Holloway, University of London

A.Johnstone@rhul.ac.uk or E.Scott@rhul.ac.uk

Abstract. We discuss generalisations of bottom up parsing, emphasising the relative costs for real programming languages. Our goal is to provide a roadmap of the available approaches in terms of their space and time performance for programming language applications, focusing mainly on GLR style algorithms. It is well known that the original Tomita GLR algorithm fails to terminate on hidden left recursion: here we analyse two approaches to correct GLR parsing (i) the modification due to Farshi that is incorporated into Visser's work and (ii) our own right-nullable GLR (RNGLR) algorithm, showing that Farshi's approach can be expensive. We also present results from our new Binary RNGLR algorithm which is asymptotically the fastest parser in this family and show that the recently reported reduction incorporated parsers can require automata that are too large to be practical on current machines.

1 Introduction

Generalisations of deterministic parsers are increasingly being applied to programming language processors: several implementations of Eiffel use Earley parsers; the most widely used recursive descent parser generators (PCCTS [1] and its successor ANTLR) use an approximate LL(k) algorithm extended with lookahead predicates; and Bison has recently acquired a GLR mode [2]. Many less widely used parser generators exist that provide various methods of backtracking, lookahead and breadth first search.

These developments might have been surprising to language designers from forty years ago: a major result from the work on deterministic top down and bottom up parsing algorithms was that near-deterministic languages are sufficiently expressive to give the human programmer a comfortable notation that, even on the very small machines of the day, allowed thousands of tokens to be parsed per second. Since then, it has been conventional to use LALR or LL techniques to implement our synthetic languages, supplemented with simple disambiguation strategies to cope with, for instance, the IF-THEN-ELSE ambiguity. The appearance of more powerful parser generators is in part a reflection of the enormous gains in processor performance and (perhaps more importantly) memory availability since the development of LALR and LL parser generators in the early 1970's when both memory sizes and instruction rates were typically a small multiple of 10^4 . Commodity computers now sport physical memory capacities and instruction rates in the 10^9 range.

Of course, when the shape of the language is not under a designer's control, we cannot assume the existence of a deterministic grammar. Natural language parsing (NLP) has to contend with nondeterministic languages that are usually ambiguous as well: a parser must thus incorporate mechanisms for finding particular derivations from a potentially infinite set. Generalised parsing has therefore always been central to NLP applications and developments of Earley style chart parsers and Tomita style generalised LR (GLR) parsers have continued in the NLP field whilst computer science based research in parsing has declined. Other applications include biological sequence analysis [3] (where context free searches can deliver fewer false positives than those specified with regular expressions where nested matching components occur) and, closer to home, algebraic specification of language processors for reverse engineering [4]. It is sometimes hard to predict how NLP style parsers will cope with programming languages. The distinguishing characteristics of NLP systems from a parsing point of view are that the grammars are large (six or seven thousand productions is not unusual) and the input sentences are usually only at most a few tens of symbols long. In contrast, the ANSI-C grammar (after expansion of optional phrases) has only 71 nonterminals and 230 productions, but source files containing thousands of tokens are normal.

This paper is about generalisations of bottom up parsing; their size and performance relative to other kinds of general parsers. We shall identify a variety of practical limitations and discuss the practical performance in terms of (i) the *size* of a parsing program and its associated tables; (ii) the run time performance; and (iii) the run time space requirements. We compare performance with Earley's parser. We have discussed elsewhere the limitations of arbitrary lookahead top down parsers [5]. Here our goal is to provide a roadmap of the available GLR approaches in terms of their space and time performance for small-grammar/long-input problems.

It is well known that the original Tomita GLR algorithm fails to terminate on hidden left recursion. We analyse two approaches to correct GLR parsing: the modification due to Farshi [6] that is incorporated into Rekers' [7] and Visser's [8] work; and our right-nullable GLR (RNGLR) algorithm [9,10]. We also present results from our new Binary RNGLR algorithm [11] which is asymptotically the fastest parser in this family. In addition, we present results on the size of parsers generated using recently reported reduction incorporated automata [12,13] which indicate that the commonly encountered grammar constructs can yield parsers that are too large to be practical on current machines.

The paper is organised as follows. For completeness, we begin by noting some of the limitations of backtracking parsers. In section 3 we distinguish between recognisers and parsers, which must return a (potentially infinite) set of derivations, and consider the size of the recognisers. Sections 4 and 5 summarise the historical development of some recognition matrix-based algorithms and GLR algorithms, respectively. Experimental results for a variety of techniques are given in section 6. Section 7 describes an algorithm whose underlying automaton contains 'reduction' edges and gives experimental results which show that

such automata are probably impractically large. We demonstrate some simple transformations that can help ameliorate this problem in some cases.

2 Backtracking and Lookahead

Many authors have described parsing algorithms in which deterministic approaches are enhanced with backtracking and lookahead. Such parsers generally have exponential performance bounds in their naïve form. Algorithms augmented with a well-formed substring table (actually, the CYK recognition matrix described below) can avoid re-computing substring matches, in which case $O(n^3)$ time bounds can be achieved although we know of no production backtracking parser generator that uses this strategy.

BtYACC (Backtracking YACC) [14] is a bottom up backtracking parser: Berkeley YACC was modified to perform exhaustive backtracking when it comes across a conflict in the table by first shifting and then backing up and making all possible reductions until it finds a successful parse or until it has tried every possibility. Using this technique it is relatively simple to produce quadratic and exponential parse-time behaviours. The general principle is to force the parser to shift past many potential reduction steps whilst pursuing fruitless paths within the grammar, thus maximising the wasted backtracking steps. Consider the (ambiguous) grammar

$$\begin{aligned} S &::= Ab \mid A'c \\ A' &::= A'a \mid a \\ A &::= aA \mid Aa \mid a \end{aligned}$$

Strings in the b -sublanguage are parsed in linear time, but exponential time is required to parse strings in the c -sublanguage. Using string lengths n from 10 to 20 tokens on a 120MHz Pentium, we found that the run times measured as above are a good fit to $6.75 \times 10^{-6} 2^n$ which translates to run times of over 7 seconds for strings of only 20 characters (and a little over 240 years for strings of 50 tokens).

For completeness we note that several public domain top down backtracking parser generators utilise a longest match strategy, returning a single match from each parser function. This approach requires a grammar to be ordered so that the longest match is encountered first by the parser. Sadly, there exist languages that are incompatible with ordered longest match in that the rules must be ordered one way for some strings and another way for others. Of course, even correct recursive descent backtracking parsers cannot cope with left recursion since the recursion will never bottom out.

3 Parsers and Recogniser Size

In the rest of this paper we focus on algorithms which attempt to explore all possible derivations in a breadth-first manner. In this section we look briefly at the additional issues surrounding derivation output and then consider the size of the recognisers themselves.

Deterministic LL and LR parsers can easily provide a derivation as a side effect of the parsing process, but in general we must distinguish between *recognisers* which simply return a boolean result and *parsers* which must return at least one derivation of a string. Recognition is inherently cheaper than parsing and some care is required when interpreting reports of the asymptotic behaviour of algorithms. Algorithms such as Earley, CYK and Valiant return recognition matrices from which a single derivation may be extracted in $O(n^2)$ time, while Tomita style algorithms (Farshi, RNGLR, BRNGLR) and Reduction Incorporated (RI) parsers return a Shared Packed Parse Forest (SPPF) which encodes all derivations.

The parser versions of Earley's algorithm return only one derivation. As noted by Tomita [15], Earley's own attempt to return all derivations contains an error. As reported by Johnson [16], any parser which returns a Tomita style SPPF will be inherently of unbounded polynomial order. The RI parser also returns a Tomita style SPPF and hence has the same order as the RNGLR parser, despite the fact that the underlying RI recogniser is cubic. The BRNGLR algorithm returns a binary form of SPPF which is of at most cubic size, and is the only general parser discussed here which returns a representation of all the derivations and which is $O(n^3)$. For the rest of this paper we consider only the results for the underlying recognisers of each algorithm.

Algorithms are usually compared in terms of their space and time requirements at run time. A parser looking at short strings may be fast and need little run time space, but if its table contains many millions of states it may still be impractical, so we must also consider the size of the parser. Table 1 summarises asymptotic costs of the algorithms mentioned above. All of the parsers require $O(n^2)$ space for the structures that they construct at parse time.

Table 1. Asymptotic costs of some general recognisers

Algorithm	Recogniser size	Recogniser time
CYK 2-form	$O(\Gamma)$	$O(n^3)$
Valiant 2-form	$O(\Gamma)$	$O(n^{2.376})$
Earley	$O(\Gamma)$	$O(n^3)$
Farshi	$O(2^{ \Gamma +\log T +\log \Gamma })$	$O(n^{k+1})$
RNGLR	$O(2^{ \Gamma +\log T +\log \Gamma })$	$O(n^{k+1})$
BRNGLR	$O(2^{ \Gamma +\log T +\log \Gamma })$	$O(n^3)$
RI	$O(2^{ \Gamma })$	$O(n^3)$

$|\Gamma|$ means the size of the grammar, that is the sum of the lengths of all productions. k is the length of the longest production. $|T|$ is the number of terminals in the grammar. n is the length of the input string. For the last four algorithms, the parser size is the size of the underlying table.

Users of YACC might be surprised at the exponential bound for the size of the tables shown above. Knuth observed that LR(1) tables could be exponen-

tially large in the number of productions and did not present his work as being practical. It turns out that even LR(0) tables are worst-case exponential, but in practice pathological grammars are not encountered: YACC users expect small tables. Could we use LR(1) tables in practice and extend tools like YACC to full LR(1) parsing power? An (uncompressed) LR(1) table for ANSI-C produced by our GTB tool contains 1796 states and 278,380 cells, which is larger than the 382 state LALR table with 59,210 cells, but only by a factor of 4.7. We should also note that comb table compression will find more opportunities for space saving on the LR(1) table, so this ratio would improve for compressed tables. This is encouraging, but languages in which nonterminals appear in many different right hand side contexts (such as COBOL) would show less favourable ratios.

4 Recognition Matrices

Initial approaches to general parsing were based on the construction of a recognition matrix M : an $n \times n$ upper triangular array each element of which contains a set of grammar slots (dotted rules, or LR(0) items). An item is in $M_{i,j}$ if it matches a substring of the input which begins at token i and extends to token j . See [17] for a very approachable unified discussion of these algorithms.

In the CYK algorithm only slots with the dot at the end of a rule are permitted, that is, a CYK parser constructs a table of complete rule matches for all substrings of the grammar. For a grammar that is in Chomsky Normal Form, the table requires $O(n^2)$ space and may be constructed in $O(n^3)$ time. A derivation (but not the complete set of derivations) may be extracted from the table in $O(n^2)$ time. It is not hard to generalise the algorithm so that it accepts arbitrary grammars, but the run time is $O(n^{k+1})$ where k is the length of the longest production.

Valiant established that the same CYK recognition matrix can be constructed using a closure operation equivalent to the multiplication of boolean matrices. This can be performed in time $O(n^{2.376})$ but the constants of proportionality are very high, making the algorithm uncompetitive for the length of inputs normally encountered in programming language applications.

The Earley algorithm also effectively constructs a recognition matrix, but this time the entries are arbitrary slots, not just complete productions. As originally described, Earley constructs n sets of ordered pairs (slot, length of substring). We give experimental results for Earley parsing in section 6.

5 Generalised LR Parsers

The fundamental problem in all generalised parsers is to simulate nondeterminism on a deterministic machine by having a single parser program examine multiple execution threads corresponding to the potentially many derivations. Backtracking parsers do this by pursuing each avenue in a depth first fashion, looking backwards and forwards in the input string as necessary. An alternative strategy is to perform a breadth first search, maintaining a list of all possible

(partial) derivations up to the current input pointer. Knuth-style LR parsers (which may be using LR(k), SLR(k) or LALR(k) tables) are an attractive starting point for breadth first search because the entire parser configuration is encoded onto a stack of state numbers. Maintaining breadth first multiple threads of control therefore requires us only to maintain multiple stacks, with the parsing algorithm modified to ensure that all active stacks are scanned for reduction actions before advancing the input pointer and performing shift actions. If we use a pointer based representation for our stacks, then stack bifurcation leads to a tree structure. Tomita noted that the set of values that can appear at the top of a stack is just the finite set of parser states, and that the context free nature of stack activations meant that branches of the tree can be merged when they have the same leaf value. He called the resulting object a Graph Structured Stack (GSS). There is now a family of algorithms (the GLR parsers) based on the GSS representation of LR parse stacks.

The general principle of a GLR parser is as follows. In an LR parser, and by extension a GLR parser, a reduction by the rule $A ::= \alpha$ corresponds to retracing edges in the LR deterministic automaton that are labelled with the elements of α , and then traversing an out edge labelled A to the ‘go to’ state g . If $|\alpha|$ is the length of α , then a reduction in a deterministic parser corresponds to simply popping $|\alpha|$ states from the (single) stack, and then pushing g . In a GLR parser, edges in the GSS must be retraced: one interpretation of the GSS is that it is simply the trace of all the LR DFA traversals made during a parse. For each path of length $|\alpha|$ found, an edge is added to the destination from state g on the current stack frontier.

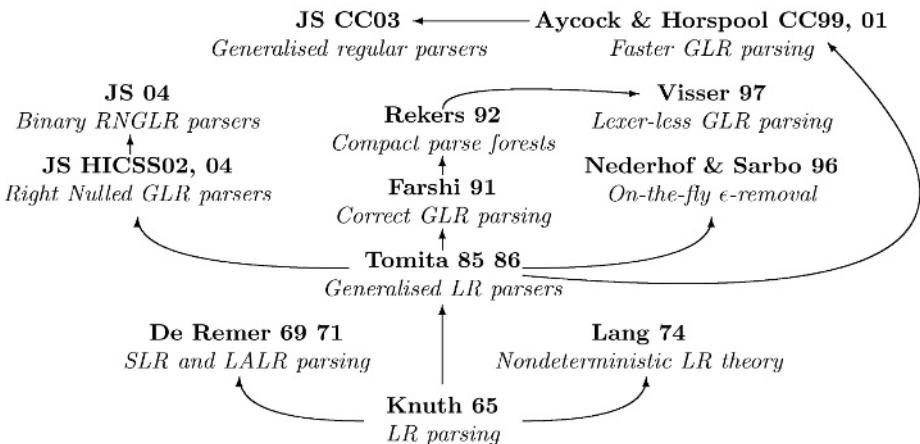


Fig. 1. The development of generalised LR parsers

Figure 1 shows the key developments in parsers which use GSS-like structures. Knuth’s seminal 1965 paper contains most of the known results concerning

LR parsers. De Remer's LALR and SLR work was the basis of practical implementations such as YACC. Lang [18] considered the theoretical opportunities for simulating nondeterminism on a sequential machine. Tomita's work can be seen as a concrete implementation of Lang's work, with the GSS structure as a primary contribution. It is not clear whether Tomita knew of Lang's work at the time. Tomita [15] actually describes five algorithms. His first algorithm is correct for ϵ -free grammars, and the subsequent versions develop extra machinery to cope with reductions by ϵ -rules. Unfortunately, there is an error which leads to non-termination on grammars with hidden left recursion. Farshi's solution [6] is to apply an exhaustive search of all states active on the current frontier for new reductions. This search can be expensive, as we shall demonstrate below. Rekers' [7] and Visser's [8] theses modify Farshi's algorithm to produce denser parse forests and to run without a lexer respectively, but the search associated with ϵ -reductions is retained. Our RNGLR [9,10] algorithm presents an elegant fix to Tomita which uses a modified form of reduction. Our BRNGLR algorithm uses a 'binarisation' of the search process to yield a Tomita style parser which is cubic on any grammar. The RI and Aycock and Horspool parsers are inspired by, but not directly related to GLR parsing. We consider them in section 7.

6 Experiments on Earley and GLR Parsers

In this paper we have focused our experimental work on four grammars. We used standard ANSI C both because it is a widely used language and because it triggers some of the unpleasant behaviour in the RI algorithm. However, the grammar for ANSI C is ϵ -free so we have also used a grammar for Pascal based on the ISO-7185 grammar. This grammar is given in EBNF so the grammar we have used has been generated from the EBNF grammar using our `ebnf2bnf` tool.

To demonstrate the non-cubic behaviour of the RNGLR and Farshi algorithms we have used the grammar Γ_1 whose rule is

$$S ::= SSS \mid SS \mid b$$

This is one of a family of grammars used by Johnson to show that Tomita's original algorithm is worse than cubic in certain cases.

To further demonstrate the differences between the algorithms we use Γ_2

$$S ::= T \quad T ::= Ab \mid TTT \quad A ::= TbAAA \mid TTb \mid \epsilon$$

The grammars Γ_1 and Γ_2 are considered in detail in [11].

Parsers for Γ_1 and Γ_2 are exercised using a string of 100 and 50 b 's respectively. Parsers for Pascal are run on tokenised versions of two source files, `quadratic` a quadratic root calculator which contains 279 tokens, and `tree viewer` a tree construction and visualisation application which has 4,425 tokens. Parsers for ANSI-C are run on a tokenised version of a Quine-McCluskey boolean minimiser containing 4,291 tokens.

We begin by comparing the total size of the Earley sets with the size of the GSS constructed by the RNGLR algorithm, and comparing the number of symbol comparisons required to construct the Earley sets with the number of edge visits required by the RNGLR algorithm to construct the GSS. As described in [11], an edge is only counted as visited if it is traversed as part of the application of a reduction. The creation of an edge is not counted as a visit.

Grammar	Input length	Earley set size	Earley symbol comparisions	RNGLR GSS nodes	RNGLR GSS edges	RNGLR edge visits
ANSI C	4,291	283,710	2,994,766	28,323	28,477	4,450
Pascal	279	7,648	53,068	1,263	1,266	364
Γ_1	100	25,355	1,628,586	398	14,949	12,405,821
Γ_2	50	15,902	905,536	578	19,737	14,726,271

We see that in general, the Earley sets are larger than the RNGLR GSS.

Next we compare the Farshi and the RNGLR algorithms which construct the same GSS so we report only the numbers of edge visits in each case. Farshi's algorithm specifies that each time an edge is added to an existing node in the GSS, for each node w not currently waiting to be processed and for each reduction of length q associated with w , all paths of length q from w which contain the new edge must be re-traced. In a naïve implementation, this means that all paths of length q must be searched in order to establish whether they include the required edge. This strategy clearly undermines the efficiency of Tomita's original approach which was designed to ensure that each path was only explored once, and in the limit will result in an explosion of the searching phase of the algorithm. To illustrate this point we have implemented the algorithm as written in [6] and reported the number of edge visits. In practice when searching for paths which contain the new edge we can terminate a search branch once it moves down a 'level' in the GSS, so we have also implemented an optimised version of Farshi's algorithm.

Grammar	Input length	Naïve Farshi edge visits	Optimised Farshi edge visits	RNGLR edge visits
ANSI C	4,291	30,754	30,484	4,450
Pascal	279	1,350	1,313	364
Pascal	4,425	23,505	22,459	5,572
Γ_1	100	2,429,702,294	15,154,234	12,405,821
Γ_2	50	3,382,571,879	15,770,426	14,729,271

As we have remarked, the worst case performance of both the Farshi and RNGLR algorithms is unbounded polynomial. In [11] we describe the BRNGLR algorithm which we prove is worst case $O(n^3)$ and has worst case $O(n^2)$ space requirements. However, the size of the GSS constructed by BRNGLR will be larger (by a small constant factor) than that constructed by the RNGLR algorithm. So in this case we report the relative GSS sizes and edge visits. As discussed in [11] the interesting case is Γ_1 on which the BRNGLR algorithm is cubic but the RNGLR and Farshi algorithms are quadratic.

Grammar	Input length	BRNGLR/RNGLR GSS nodes	BRNGLR/RNGLR GSS edges	BRNGLR/RNGLR GSS edge visits
ANSI C	4,291	30,331/28,323	30,512/28,477	4,446/4,450
Pascal	279	1,424/1,263	1,428/1,266	364/364
Pascal	4,425	23,521/21,039	23,655/21,135	5,570/5,572
Γ_1	100	496/398	29,209/14,949	1,407,476/12,405,821
Γ_2	50	764/578	32,078/19,737	448,899/14,726,271

7 Reduction Incorporated Parsers

Aycock and Horspool [12,19] described a novel parser in which the cost of a reduction is reduced in some cases. However, their algorithm can only be used on grammars without hidden left recursion. Our RI parser [13] is very closely related to their work, but is truly general. These parsers use a GSS-style structure to represent the multiple threads of control within the underlying automaton, however RI parsers do not use LR tables and do not behave like LR(1) parsers on LR(1) grammars, so we prefer to think of them as a separate family.

The motivation behind Aycock and Horspool's algorithm was to lower the constant of proportionality in Tomita style algorithms by reducing the amount of stack activity generated. Aycock and Horspool have reported an improvement in execution time for certain ambiguous grammars but to date there has been no published data on the size of the parser itself.

LR parsers are based on a DFA which recognises precisely the *viable strings* [20] of the input grammar. In the RI approach, for a grammar Γ with start symbol S , a new grammar Γ_S is constructed by ‘terminalising’ all but the non-hidden left recursive instances of all nonterminals. The (finite) set of viable strings is then generated and built into a trie, to which ‘reduction’ transitions are added. The result is a *reduction incorporated automaton* $RIA(\Gamma_S)$ which accepts precisely the language of Γ_S . Then, for each nonterminal A in Γ which was terminalised, an automaton $RIA(\Gamma_A)$ is constructed from the grammar obtained from Γ by taking A to be the start symbol. Finally an automaton is constructed by ‘stitching together’ the RIA’s as follows. Each transition, in any $RIA(\Gamma_A)$, from a state h to a state k , say, labelled with a terminalised B is replaced with a transition labelled k from h to the start state of $RIA(\Gamma_B)$. We call the resulting push down automaton a *recursive call automaton* for Γ , $RCA(\Gamma)$. We have shown, [21], that $RCA(\Gamma)$ accepts precisely the language of Γ .

In order to construct the tries it is necessary to remove all but non-hidden left recursion from the grammar. We have given [21] a different method which allows for the construction of a nondeterministic automaton, called $IRIA(\Gamma)$, from any given grammar. We terminalise the grammar so that the only recursion is non-hidden left and right recursion, then build $IRIA(\Gamma_A)$ for nonterminals A . Finally, we use the standard subset construction algorithm to generate $RIA(\Gamma_A)$ from $IRIA(\Gamma_A)$ and build $RCA(\Gamma)$ as above. We have also provided an RCA traversal algorithm which terminates and is correct even for grammars with hidden left recursion. This algorithm is the RI algorithm which is a worst case

cubic recogniser for all context free grammars, and which has less associated stack activity than the RNGLR algorithm.

The Size of Reduction-Incorporated Automata

The speed up of the RI algorithm over the RNGLR algorithm is obtained by effectively unrolling the grammar. In a way this is essentially a back substitution of alternates for instances of nonterminals on the right hand sides of the grammar rules. Of course, in the case of recursive nonterminals the back substitution process does not terminate, which is why such instances are terminalised before the process begins. However, even without recursion we would expect full back substitution to generate an explosion in the number and size of the grammar alternates. It is often possible to show artificial pathological examples which generate disproportionately large objects, just as it is possible to generate exponentially large LR(0) tables. However, the generation of large RCA's does not require artificial examples.

We terminalised and then built the RIAs and RCAs for the Pascal, ANSI C, and the Γ_1 and Γ_2 grammars described above. In each case the grammars were terminalised so that all but non-hidden left recursion was removed before constructing the RIAs. Figure 2 shows the number of terminalised nonterminals, the number of instances of these terminalised nonterminals in the grammar, the number of symbol labelled transitions in $RCA(\Gamma)$ and the number of reduction labelled transitions in $RCA(\Gamma)$. (It is a property of $RCA(\Gamma)$ that the number of states is equal to the number of symbol labelled transitions + the number of terminalised nonterminal instances + 1.)

Table 2. The size of some reduction incorporated automata

Grammar	Terminalised nonterminals	Instances of terminalised nonterminals	Symbol transitions	Reduction transitions
Pascal	8	11	13,522	11,403
ANSI C	12	42	6,907,535	5,230,022
Γ_1	1	3	5	4
Γ_2	2	6	34	19

The basic problem is that each terminalised instance of a nonterminal B in the grammar generates a sub-automaton in $RIA(\Gamma_S)$ of the size of $RIA(\Gamma_B)$. We say that a nonterminal B *depends* on A if there is a rule $B ::= \alpha A \beta$ and that a *dependency chain* is a sequence of distinct nonterminals B_0, B_1, \dots, B_n such that B_i depends on B_{i+1} , $0 \leq i \leq n-1$. Then the number of copies of $RIA(\Gamma_{B_n})$ contributed to $RIA(\Gamma_S)$ by this chain is $c_1 \times c_2 \times \dots \times c_n$ where c_i is the number of instances of B_{i+1} in the rules for B_i . Thus by constructing a grammar with n nonterminals each of which has two instances of the next nonterminal in its rules

and at least one terminal we can construct a grammar of size $O(n)$ which has an RIA of size at least $O(2^{n+2})$. If we construct a grammar with n nonterminals each of which has k instances of the next nonterminal in its rules we have a grammar of size $O(n)$ which has an RIA of size at least $O(2k^{n+1})$. We have constructed the RIA's for the family of grammars, *Family*₁,

$$B_0 ::= B_1 \mid a_1 B_1 \quad B_1 ::= B_2 \mid a_2 B_2 \quad \dots \quad B_{n-1} ::= B_n \mid a_n B_n \quad B_n ::= a$$

with $n = 2, 5, 10, 16$. (Our implementation was unable to cope with $n = 20$.) Note that there is no recursion in the grammars and $RCA(\Gamma) = RIA(\Gamma_{B_0})$. The sizes of the RIAs are given in the following table.

n	2	5	10	16
symbol transitions	15	127	4,095	262,144
reduction transitions	11	95	3,071	196,607

The potential explosion in automaton size is not of itself surprising: we know it is possible to generate an LR(0) automaton which is exponential in the size of the grammar. The point is that the example here is not artificial. In the ANSI C grammar the expression subtree contains a non-recursive dependency chain of depth 16, and five nonterminals in the chain contain at least three instances of the next nonterminal on the right hand sides of their rules. It is this that causes the RCA for ANSI C to be impractically large: the total number of edges in the automaton is 12,137,557.

In the above case we could right factor the grammar, replacing each rule $B_{i-1} ::= B_i \mid a_i B_i$ with two rules $B_{i-1} ::= C_i B_i$, $C_i ::= \epsilon \mid a_i$. This generates the following results

n	2	5	10	16	20
symbol transitions	9	18	33	51	63
reduction transitions	8	17	32	50	62

However, this approach can not be applied to grammars which can not be right factored in this way, for example the family *Family*₂

$$B_0 ::= B_1 \mid a_1 B_1 b_1 \quad B_1 ::= B_2 \mid a_2 B_2 b_2 \quad \dots \quad B_{n-1} ::= B_n \mid a_n B_n b_n \quad B_n ::= a$$

Further, the point of a general parsing technique is that it can be used without modification to the grammar, allowing the grammar to be written in a way which is natural for the required semantics.

So far we have only discussed the size of the final automaton $RCA(\Gamma)$. We also need to worry about the space required to construct the parser. The construction method that we have described [13] produces intermediate automata, $IRIA(\Gamma_A)$, to which the subset construction is then applied to generate the trie-style automata $RIA(\Gamma)$. The $IRIA(\Gamma_A)$ do not have the merging of paths on common left hand ends which is a feature of the $RIA(\Gamma_A)$ thus the explosion

described above will also occur in $IRIA(\Gamma_{B_0})$ (but not in $RIA(\Gamma_{B_0})$) for the grammar family *Family*₃

$$B_0 ::= B_1 \mid B_1 a_1 \quad B_1 ::= B_2 \mid B_2 a_2 \quad \dots \quad B_{n-1} ::= B_n \mid B_n a_n \quad B_n ::= a$$

Perhaps more importantly, the same effect is triggered by left recursive rules. For a rule in Γ of the form

$$A ::= A\beta \mid \alpha$$

$IRIA(\Gamma_A)$ contains a sub-automaton equivalent to an automaton $IRIA(\Gamma_{A'})$ generated by the rule $A' ::= \alpha$. The recursive call to A implies that $IRIA(\Gamma_A)$ effectively contains two copies of $IRIA(\Gamma_{A'})$. The effect becomes explosive when there is more than one left recursive rule

$$A ::= A\beta \mid A\gamma \mid A\delta \mid \alpha$$

In this case $A' ::= A\gamma \mid A\delta \mid \alpha$ and so $IRIA(\Gamma_{A'})$ contains two copies of $IRIA(\Gamma_{A''})$ and of $IRIA(\Gamma_{A'''})$ where $A'' ::= A\delta \mid \alpha$ and $A'''' ::= A\gamma \mid \alpha$.

Again this is not an artificial example. In the ANSI C grammar one nonterminal, `postfix_expression`, in the expression subtree has seven alternates, six of which begin with a recursive call to itself.

All these sub-automata will be merged by the subset algorithm and hence do not cause an explosion in the size of the RIA, but using the method described in [13] they must be constructed. Thus we have modified our IRIA construction method slightly so that, as for the trie construction method, some aspects of the subset construction are carried out ‘on-the-fly’.

IRIA Representation

The IRIA are not held in memory in their full form. There are *headers* representing each instance of each nonterminal in the terminalised grammar Γ_S , and these headers effectively contain links into the grammar Γ_S in such a way that the final structure can be traversed as though it were $IRIA(\Gamma_S)$. This means that for practical purposes the size of $IRIA(\Gamma_S)$ is the number of headers plus the size of Γ . The construction method described in [13] requires a new header for each direct left recursive call of a nonterminal to itself, but in fact the original header can be re-used in this case because the same automaton will be constructed either way by the subset construction. Thus we have modified the IRIA construction algorithm to re-use the original header in the case where a nonterminal instance is the result of direct left recursion.

Using this we find that the IRAs have a size which is comparable with the corresponding RIAs. For example, for the Pascal and ANSI C grammars we have

grammar	total IRIA headers	RIA symbol transitions	RIA reduction transitions
Pascal	7,759	13,522	11,403
ANSI C	9,594,266	6,907,535	5,230,022

For both Pascal and ANSI C the numbers of headers required for each of the sub IRIA is between half and two thirds of the number of symbol transitions in the corresponding RIA. The increase in the total number of headers required for ANSI C is because the `conditional_expression` IRIA, whose size dominates the total size, contains some non-left recursive rules which could be left factored, i.e. rules of the form $A ::= B \mid BC$.

We have further modified our construction procedure so that it effectively works on grammars written in a form of EBNF. The input grammar supplied by the user is still in BNF form, and the output derivation trees are still with respect to this grammar. But internally the IRIA construction procedure uses the grammar in a factored form: $A ::= B(\epsilon \mid C)$. This has the effect of ensuring that the size of the IRAs is comparable with the size of the final RIA. We could extend this approach further and use, internally, right factored grammars but the problem then is dealing correctly with the reduction transitions. We return briefly to this below, when we discuss the issues surrounding derivation tree generation.

Controlling the Size of the RCA

We conclude this discussion by observing that ultimately the way to control the explosion in the size of $RCA(\Gamma)$ is to introduce more nonterminal terminalisations. We must terminalise the grammar in such a way that Γ_S has no self embedding for the process to be correct, but the process is still correct if further terminalisations are applied. By terminalising the middle nonterminal in a dependency chain we will effectively replace the single automaton whose size is $O(2^{n+2})$ with two automata $RIA(\Gamma_{B_0})$ and $RIA(\Gamma_{B_{n/2}})$ each of size $O(2^{(n+1)/2})$. If we take this approach with the family *Family*₁ of grammars above and terminalise by simply replacing the rule for $B_{n/2}$ with

$$B_{(n/2)-1} ::= (B_{n/2})^\perp \mid a_{n/2}(B_{n/2})^\perp$$

and leaving the other rules the same (we assume for simplicity that n is even) we get the following results

n	2	6	10	16	20
symbol transitions	12	54	222	1,790	7,678
reduction transitions	8	38	158	1,278	5,118

Here we have reported the total numbers of transitions in the two RIA automata created in each case.

Taking this approach with the ANSI C grammar, terminalising non-recursive instances of a nonterminal in the middle of the expression sub-grammar, we get the following results

Grammar	Terminalised nonterminals	Instances of terminalised nonterminals	Symbol transitions	Reduction transitions
ANSI C	13	47	21,895	16,573

Of course, this approach will also work for *Family*₂ and can significantly reduce the parser size for any grammar with long dependency chains. However, it is at the cost of introducing more recursive calls when the parser is run and hence the (practical but not asymptotic) run time performance and space requirements of the parser will be increased. Thus there is an engineering tradeoff to be made between the size of the parser and its performance.

As we have remarked above, the parser version of the RI algorithm is worst case unbounded polynomial. We expect to be able to modify our SPPF construction so that it produces binary SPPFs in a form similar to those produced by the BRNGLR algorithm by internally treating a grammar rule of the form $A ::= XYZW$ as though it were an EBNF rule of the form $A ::= ((XY)Z)W$. This approach will automatically produce smaller parsers (RCAs) than those currently produced by the Aycock and Horspool approach in the case where the grammar can be right factored. This will also permit the grammar right-factoring mentioned above, but this is still work in progress.

8 Conclusions

We conclude that Earley and GLR style parsers are practical on current machines for programming language applications. Of these, the BRNGLR algorithm is asymptotically fastest but requires slightly more space than the polynomial RNGLR algorithm. RNGLR performs fewer GSS traversals than Farshi's (and by extension Visser's) algorithms. The original Tomita algorithm fails to terminate on hidden left recursion, and in any case constructs larger GSS's as a result of sub-frontier creation. Reduction Incorporated algorithms at present strain the limits of practicality even for standard programming languages. The underlying automaton for the RI parser for ANSI-C contains over 12×10^6 edges. However RI algorithms may become generally useful in the future and do have some present applications. An engineering tradeoff is possible: they can be made practical by introducing additional sub-automaton calls effectively trading execution time for parser size.

References

1. Parr, T.J.: Language translation using PCCTS and C++. Automata Publishing Company (1996)
2. Eggert, P.: <http://compilers.iecc.com/comparch/article/03-01-042>. (2003)
3. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological Sequence Analysis. Cambridge University Press (1998)
4. van den Brand, M., Heering, J., Klint, P., Olivier, P.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems **24** (2002) 334–368
5. Johnstone, A., Scott, E.: Generalised recursive descent parsing and follow determinism. In Koskimies, K., ed.: Proc. 7th Intnl. Conf. Compiler Construction (CC'98), Lecture notes in Computer Science 1383, Berlin, Springer (1998) 16–30

6. Nozohoor-Farshi, R.: GLR parsing for ϵ -grammars. In Tomita, M., ed.: Generalized LR parsing. Kluwer Academic Publishers, Netherlands (1991) 60–75
7. Rekers, J.G.: Parser generation for interactive environments. PhD thesis, University of Amsterdam (1992)
8. Visser, E.: Syntax definition for language prototyping. PhD thesis, University of Amsterdam (1997)
9. Scott, E., Johnstone, A.: Reducing non-determinism in reduction modified GLR parsers. To appear in: *Acta Informatica* (2004)
10. Johnstone, A., Scott, E.: Generalised reduction modified LR parsing for domain specific language prototyping. In: Proc. 35th Annual Hawaii International Conference On System Sciences (HICSS02). IEEE Computer Society, IEEE, New Jersey (2002)
11. Scott, E., Johnstone, A., Economopoulos, G.: BRN-table based GLR parsers. Technical Report TR-03-06, Royal Holloway, University of London, Computer Science Department (2003)
12. Aycock, J., Horspool, N.: Faster generalised LR parsing. In: Compiler Construction: 8th International Conference, CC'99. Volume 1575 of Lecture Notes in computer science., Springer-Verlag (1999) 32 – 46
13. Johnstone, A., Scott, E.: Generalised regular parsers. In Hedin, G., ed.: Compiler Construction, 12th Intnl. Conf, CC2003. Volume 2622 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2003) 232–246
14. Dodd, C., Maslov, V.: <http://www.siber.com/btyacc>. (2002)
15. Tomita, M.: Efficient parsing for natural language. Kluwer Academic Publishers, Boston (1986)
16. Johnson, M.: The computational complexity of GLR parsing. In Tomita, M., ed.: Generalized LR parsing. Kluwer Academic Publishers, Netherlands (1991) 35–42
17. Graham, S.L., Harrison, M.A.: Parsing of general context-free languages. *Advances in Computing* **14** (1976) 77–185
18. Lang, B.: Deterministic techniques for efficient non-deterministic parsers. In: Automata, Languages and Programming: 2nd Colloquium. Lecture Notes in computer science, Springer-Verlag (1974) 255 – 269
19. Aycock, J., Horspool, R.N., Janousek, J., Melichar, B.: Even faster generalised LR parsing. *Acta Informatica* **37** (2001) 633–651
20. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles techniques and tools. Addison-Wesley (1986)
21. Scott, E., Johnstone, A.: Table based parsers with reduced stack activity. Technical Report TR-02-08, Royal Holloway, University of London, Computer Science Department (2002)

An Automata-Theoretic Algorithm for Counting Solutions to Presburger Formulas

Erin Parker^{*1} and Siddhartha Chatterjee²

¹ Department of Computer Science, University of North Carolina,
Chapel Hill, NC 27599, USA, parker@cs.unc.edu

² IBM T. J. Watson Research Center, 1101 Kitchawan Road,
Yorktown Heights, NY 10598, USA, sc@us.ibm.com

Abstract. We present an algorithm for counting the number of integer solutions to selected free variables of a Presburger formula. We represent the Presburger formula as a deterministic finite automaton (DFA) whose accepting paths encode the standard binary representations of satisfying free variable values. We count the number of accepting paths in such a DFA to obtain the number of solutions without enumerating the actual solutions. We demonstrate our algorithm on a suite of eight problems to show that it is universal, robust, fast, and scalable.

1 Introduction

The problem of counting integer solutions to selected free variables of Presburger formulas is relevant to several applications in the analysis and transformation of scientific programs. Researchers have used Presburger formulas to model the iterations of a loop nest [1], the memory locations touched by a loop nest [2], the cache lines touched by a loop nest [2], and the memory accesses incurring cache misses [3]. The well-known Cache Miss Equations (CME) compiler framework [4] for analyzing and tuning memory behavior is based on Presburger arithmetic. Counting the number of solutions to such formulas allows us to estimate the execution time of a loop nest, to evaluate the message traffic generated by a loop nest, and to determine the cache miss rate of a loop nest, among other things. However, the counting problem is inherently difficult to solve in its full generality. Early work [2,5,6,7,8] therefore took *ad hoc* approaches to this counting problem; more recent work by William Pugh [9] and Philippe Clauss [1] provided solution methods that are more systematic and complete.

Pugh's method [9] is based on computing sums. He introduces techniques for summing a polynomial over all integer solutions of selected free variables of a Presburger formula. He expresses the number of solutions symbolically, in terms of symbolic constants that are the remaining free variables in the Presburger formula. For a given summation problem, the choice of which techniques to apply and the order in which to apply them seems to require human intelligence. To our knowledge, no software implementation of Pugh's method exists.

^{*} Erin Parker is supported by a DOE High-Performance Computer Science Fellowship.

Clauss' method [1] is based on the vertices of parameterized polytopes and Ehrhart (pseudo-)polynomials [10]. He models a Presburger formula as a disjoint union of rational convex polytopes, the number of integer points in which is equal to the number of integer solutions to selected free variables in the formula. Clauss' method requires a geometric preprocessing step to express the Presburger formulas as disjoint polyhedra. Clauss expresses the number of included integer points using Ehrhart polynomials and periodic numbers, where the free variables are the size parameters (or the remaining free variables in the formula). Like Pugh, Clauss produces a symbolic expression of the number of solutions. Clauss' method is implemented in the Polylib library [11].

The examples that Pugh [9] and Clauss [1] handle have two characteristics: they are smooth functions of their input arguments, and the constants and coefficients that they contain are small (typically order 10–100). Our interest in Presburger arithmetic arises from modeling cache misses, for which the formulas are distinctly non-smooth (see Fricker *et al.* [12] for examples) and contain large constants and coefficients (cache capacity and array starting addresses being typical examples). The methods of Pugh and Clauss are impractical for handling such formulas. Pugh's method splinters the domain of a non-smooth formula into many subdomains, and the choice of the applicable techniques and their order of application grows combinatorially. Clauss' method requires an Ehrhart polynomial of very high degree and an impractically large number of coefficients, and is also subject to geometric degeneracies for many problems of interest.

Ghosh *et al.* make similar observations in their CME work commenting that “... to make the framework more effective, we should be able to automatically solve or at least find the number of solutions to the CMEs in parametric form” [4]. They mention the methods of Pugh and Clauss as possibilities, but, given the intrinsic difficulty of the counting problem, do not use these methods to find the number of cache misses explicitly. Instead, they use the CMEs indirectly to derive problem-specific optimization algorithms, and compromise the generality of their framework as a result.

Because of the insufficiency of existing methods for our purposes, we considered other general approaches for counting solutions to Presburger formulas that would work well for non-smooth formulas. Our counting algorithm is fundamentally different from the methods of Pugh and Clauss: it builds on a deep connection between Presburger arithmetic and automata theory, and then converts the counting problem for solutions of the Presburger formula into a graph-theoretic path counting problem on the related deterministic finite automaton (DFA). The key difference between our algorithm and the methods of Pugh and Clauss lies in symbolic capabilities. Both Pugh and Clauss express the number of solutions as closed-form functions of symbolic constants. *Our algorithm is not symbolic.* It produces the number of solutions for a particular set of numerical values of the symbolic constants, *i.e.*, the result of evaluating Pugh's or Clauss' function at a particular point. Whether this lack of symbolic capabilities is a limitation depends on the application. For instance, scheduling applications require closed-form formulas, while actual numbers may be sufficient for load balancing.

Our counting algorithm is not the only non-symbolic approach for determining the number of integer solutions to Presburger formulas. Barvinok's algorithm [13] (with subsequent improvements by Dyer and Kannan [14]) counts the number of integer points inside a convex polyhedron of fixed dimension in polynomial time. The LattE (Lattice point Enumeration) tool [15], the first known implementation of Barvinok's algorithm, is software for the enumeration of all lattice points inside a rational convex polyhedron. To count the number of solutions to a Presburger formula, Barvinok's algorithm requires a geometric preprocessing step to express the formula as a disjoint union of polyhedra. The LattE tool does not implement the preprocessing step. Our algorithm requires no such geometric preprocessing technique. The quite recent availability of the LattE software combined with the problem of geometric preprocessing has prevented a direct comparison with our counting approach.

The following points summarize the strengths and weaknesses of our counting algorithm.

- The algorithm is *universal*, handling any problem that is expressible in Presburger arithmetic.
- The algorithm is *robust*, handling an arbitrary representation of a problem without need for geometric preprocessing.
- The algorithm is *implemented*.
- The algorithm is *fast*, as we will demonstrate in Sect. 5.
- The algorithm can handle formulas with *large coefficients and constants*, as indicated by the example problems in Sect. 5.
- The algorithm does not handle *symbolic* constants.

The remainder of this paper is structured as follows. Section 2 provides a brief review of Presburger arithmetic. Section 3 establishes the connection between Presburger arithmetic and DFAs. Section 4 presents our counting algorithm. Section 5 demonstrates the algorithm on three examples. Section 6 concludes.

2 Presburger Arithmetic

Presburger arithmetic is the first-order theory of natural numbers with addition. A Presburger formula consists of affine equality and/or inequality constraints connected via the logical operators \neg (not), \wedge (and), \vee (or), and the quantifiers \forall (for all) and \exists (there exists). It is well-known how to express certain non-linear constraints in Presburger arithmetic, such as floors, ceilings, quotients, remainders [9], and bit-interleaving [3].

Consider Presburger formula $P(i, j, k; n) = 1 \leq i \leq n \wedge 3 \leq j \leq i \wedge j \leq k \leq 5$ with free integer variables i , j , and k and symbolic constant n ,¹ and consider the sum $S(n) = \sum_{i=1}^n \sum_{j=3}^i \sum_{k=j}^5 1$ with symbolic constant n . We say that

¹ Although i , j , k , and n are all free variables of the formula in the terminology of logic, we distinguish between those free variables that appear as summation indices in the corresponding sum and those that do not.

formula $P(i, j, k; n)$ “describes” the sum $S(n)$ because, for any given value of n , the number of solutions to the Presburger formula is equal to the value of the sum, *i.e.*, $\forall n : S(n) = |\{(i, j, k) : P(i, j, k; n) = \text{true}\}|$.

Presburger arithmetic is decidable [16]; however, the complexity of the decision procedure is superexponential in the worst case. For a sentence of length n , the bound on storage and time required is $2^{2^{pn}}$, for some constant $p > 1$ [17]. This bound is tight [18]. However, it is a worst-case bound, and prior use of Presburger arithmetic in program analysis contexts has generally been well-behaved in terms of complexity of decision and simplification algorithms.

3 The Presburger Arithmetic–DFA Connection

Our counting method exploits a fundamental connection between Presburger arithmetic and automata theory, namely, that there exists a DFA recognizing the positional binary representation of the solutions of any Presburger formula. Following standard terminology, we define a DFA M as a 5-tuple $(S, \Sigma, \delta, q_0, F)$, where S is a finite set of states, Σ is a finite set of symbols called the alphabet, $\delta : S \times \Sigma \rightarrow S$ is the transition function, $q_0 \in S$ is the start state, and $F \subseteq S$ is a set of final states. This connection is perhaps not surprising, given that DFAs can describe arithmetic on the binary representation of natural numbers. The key point to remember in transitioning from Presburger arithmetic to DFAs is that we are moving from a domain of values (natural numbers) to a domain of representations (positional binary encoding).

Büchi [19,20] originally proved that a subset of $(\{0, 1\}^n)^*$ is recognizable by a finite state automaton if and only if it is definable in WS1S (Weak Second-order Theory of One Successor). Boudet and Comon [21] build on this result. Because Presburger arithmetic can be embedded in WS1S, there is a DFA recognizing the solutions of a Presburger formula. Boudet and Comon formalize the connection between Presburger arithmetic and automata in the following theorem.

Theorem 1 (Boudet–Comon [21]). *Let $\phi \equiv Q_n x_n, \dots, Q_1 x_1 \psi$ be a formula of Presburger arithmetic where Q_i is either \exists or \forall and ψ is an unquantified formula with variables $x_1, \dots, x_n, y_1, \dots, y_m$. There is a deterministic and complete automaton recognizing the solutions of ϕ with at most $O(2^{2^{|Q|\phi|}})$ states, where $|\phi| = K(\phi) + V(\phi)$, $K(\phi)$ being the sum of the sizes of each constant in the formula ϕ , written in binary representation, and $V(\phi)$ being the number of variables in formula ϕ .*

The proof of the theorem is constructive, with the construction procedure defined by induction on the structure of ϕ . The base cases are linear equalities and inequalities, for which DFA recognizers are easy to construct [21,22,23]. Logical connectives of subformulas utilize closure properties of regular sets under intersection, union, and complementation [24]. Existential quantification is handled by projecting the alphabet and the transition function (producing a nondeterministic finite automaton) followed by determinization and state minimization. Universal quantification exploits the tautology $\forall x\phi \equiv \neg\exists x\neg\phi$.

Note that moving from Presburger formulas to DFAs does not circumvent the difficulty of counting solutions, since both the decidability of a Presburger formula and the construction of a DFA recognizing the formula have triple exponential worst-case complexity. We choose to exploit the Presburger Arithmetic-DFA connection because the complexity manifests in a better understood and manageable form. Specifically, it is the translation of universal quantifiers and negation that can cause an exponential blowup in the number of states.

The connection between Presburger arithmetic and DFAs has been widely studied in the context of verification of system properties such as safety and liveness. Several authors [22,23,25,26,27] have refined the procedure for constructing automata from Presburger formulas, extending the scope of formulas from natural numbers to integers, allowing the mixing of integer and real variables, and providing tighter bounds on the number of states of the resulting DFA. In particular, Bartzis and Bultan [22,23] present algorithms for constructing finite automata that represent integer sets satisfying linear constraints. They use MONA [28], an automata manipulation tool, to implement these algorithms. Compared to similar approaches for automata representation, the methods of Bartzis and Bultan give tighter bounds on the size of generated automata. We use the Bartzis and Bultan construction algorithms to represent an affine equality/inequality constraint as a deterministic finite automaton. Using the automata-theoretic operations of union, intersection, complementation, and projection (offered in the MONA tool’s automata package), we then combine such DFA representations of the affine equality/inequality constraints constituting a Presburger formula to get the DFA representation of the formula.

4 The Counting Algorithm

We now present our algorithm for counting solutions to a Presburger formula. Section 4.1 offers an example DFA used to illustrate our algorithm. Section 4.2 describes how a DFA represents the solutions of a Presburger formula. Section 4.3 looks at treating the DFA as a directed, edge-weighted graph. Section 4.4 explains how path length is used to ensure a correct solution count. Section 4.5 presents our algorithm for counting accepting paths in a DFA.²

4.1 Example DFA

For illustration of our counting algorithm, consider the following running example. The Presburger formula shown in Fig. 1 describes *interior misses*³ incurred by loop nest **L** (of Fig. 2.a). Figure 2.b shows a version of this formula simplified using the Omega Calculator [30,31]. Notice that the clauses in Fig. 2.b are not disjoint. However, this potential difficulty is not an issue during the construction of a DFA to represent the formula. In fact, the formula requires no geometric preprocessing for DFA construction, which makes our algorithm very robust.

² For the proofs of all lemmas and theorems in this section, see [29].

³ An interior miss is a particular type of cache miss introduced by Chatterjee *et al.* [3].

$$\begin{aligned}
& \exists d : 0 \leq i, j, k < 20 \wedge ((k = 0 \wedge R = 0 \wedge 4(128d) \leq 800 + i + 20j < 4(128d + 1)) \vee (R = 1 \wedge \\
& 4(128d) \leq i + 20k < 4(128d + 1)) \vee (R = 2 \wedge 4(128d) \leq 400 + k + 20j < 4(128d + 1)) \vee (k = 19 \wedge \\
& R = 3 \wedge 4(128d) \leq 800 + i + 20j < 4(128d + 1))) \wedge (\exists u, v, w, F, e : 0 \leq u, v, w < 20 \wedge (u < i \vee (u = i \\
& \wedge v < j) \vee (u = i \wedge v = j \wedge w < k) \vee (u = i \wedge v = j \wedge w = k \wedge F < R)) \wedge ((w = 0 \wedge F = 0 \\
& \wedge 4(128e) \leq 800 + u + 20v < 4(128e + 1)) \vee (F = 1 \wedge 4(128e) \leq u + 20w < 4(128e + 1)) \vee (F = 2 \wedge \\
& 4(128e) \leq 400 + u + 20v < 4(128e + 1)) \vee (w = 19 \wedge F = 3 \wedge 4(128e) \leq 800 + u + 20v < 4(128e + 1))) \\
& \wedge (\neg(\exists x, y, z, G : 0 \leq x, y, z < 20 \wedge (x < i \vee (x = i \wedge y < j) \vee (x = i \wedge y = j \wedge z < k) \vee \\
& (x = i \wedge y = j \wedge z = k \wedge G < R)) \wedge (u < x \wedge v < y) \vee (u = x \wedge v = y \wedge w < z) \vee \\
& (u = x \wedge v = y \wedge w = z \wedge F < G)) \wedge ((z = 0 \wedge G = 0 \wedge 4(128d) \leq 800 + x + 20y < 4(128d + 1)) \\
& \vee (G = 1 \wedge 4(128d) \leq x + 20z < 4(128d + 1)) \vee (G = 2 \wedge 4(128d) \leq 400 + z + 20y < 4(128d + 1)) \vee \\
& (z = 19 \wedge G = 3 \wedge 4(128d) \leq 800 + x + 20y < 4(128d + 1)))) \wedge \neg(d = e))
\end{aligned}$$

Fig. 1. Presburger formula describing interior misses incurred by loop nest **L** in cache set 0 (assuming a direct-mapped cache with block size 32 bytes and capacity 4K bytes, double-precision arrays adjacent in memory and linearized in column-major order).

```

L: do i = 0, 19
    do j = 0, 19
        c = Z[i,j]
        do k = 0, 19
            c = X[i,k]*Y[k,j]+c
        enddo
        Z[i,j] = c
    enddo
enddo
a.                                     b.

```

Fig. 2. a. Loop nest **L** performs the matrix multiplication $Z = X \cdot Y$. b. Simplified version of the Presburger formula in Fig. 1.

The DFA in Fig. 3 recognizes the solutions of the Presburger formula in Fig. 2.b. The start state of the DFA, q_0 , is at node 1. The final states in set F are denoted with double circles. For this DFA there is one final state at node 10. Node 2 is a garbage state. A DFA can represent n -tuples of integers in binary notation as words over the alphabet $\{0, 1\}^n$ simply by *stacking* them, using an equal length representation for each integer in the tuple. A label on any edge of the DFA in Fig. 3 is a stack of four digits, each corresponding to a free variable (i, j, k , or R). Therefore, the alphabet Σ of the DFA is

$$\left\{ \begin{array}{cccccccccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{array} \right\}.$$

Borrowing terminology from hardware logic design, an X in a stack signifies that either a 0 or 1 is possible in that position.

4.2 Encoding Free Variable Values

Given the DFA representation of a Presburger formula, each accepting path in the DFA encodes free variable values that constitute a solution to the formula. The encoding is the standard binary representation of the integer values of the

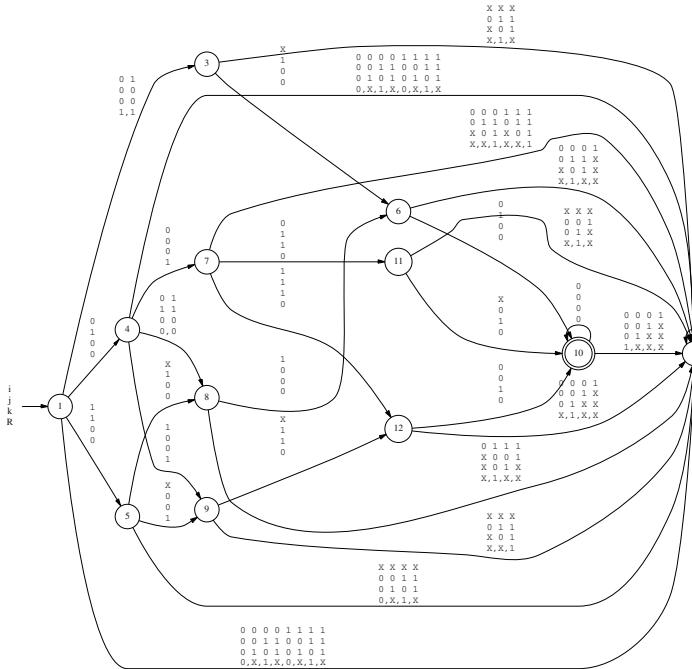


Fig. 3. DFA recognizing the solutions of the Presburger formula in Fig. 2.b.

free variables, proceeding from least significant bit (LSB) to most significant bit (MSB). Formally, an encoding of a non-negative integer b is a word $b_m \dots b_0$ such that each b_i is 0 or 1 and $b = \sum_{i=0}^m b_i 2^i$. A tuple of non-negative integers is encoded by stacking their binary representations and padding with leading 0s to make the lengths identical.

Let a path of length i from state p to state q , denoted $P_i(p, q)$, be a string $A \in \Sigma^i$ such that $\delta^i(p, A^R) = q$, where string A^R is the reversal of string A . (The string reversal is a notational device to resolve the mismatch between the conventional MSB-first representation of numbers and the LSB-first consumption of the encoding by the DFA.) In the DFA of Fig. 3, one path from the state 1 to 3, $P_1(1, 3)$, is $\begin{smallmatrix} 0 \\ 0 \\ 1 \end{smallmatrix}$. Path $P_4(1, 10) = \begin{smallmatrix} 1000 \\ 0101 \\ 1100 \\ 0010 \end{smallmatrix}$, which goes through states 4, 7, and 11, is an accepting path identifying $i = 8$, $j = 5$, $k = 12$, and $R = 2$ as a solution.

To easily identify one set of states reachable from another set of states, we extend the transition function $\delta : S \times \Sigma \rightarrow S$ to $\Delta : 2^S \rightarrow 2^S$, defined by $\Delta(S') = \bigcup_{p \in S'} \{\delta(p, a) : a \in \Sigma\}$. For the DFA in Fig. 3, $\Delta(\{6, 10, 11, 12\}) = \{2, 10\}$.

4.3 Treating the DFA as a Graph

The key to our algorithm for counting accepting paths in a DFA is to treat the DFA as a weighted, directed graph. For states p and q , let $wt(p, q) = |\{a \in \Sigma :$

$\delta(p, a) = q\}$ be the number of alphabet symbols that cause transition from state p to state q . Given DFA $M = (S, \Sigma, \delta, q_0, F)$, define a directed, edge-weighted graph as $G(M) = (V, E, W) = (S, \{(p, q) | \exists a \in \Sigma : \delta(p, a) = q\}, \lambda(p, q).wt(p, q))$.

The problem of counting the number of solutions represented by a DFA M reduces to a path-counting problem on the graph $G(M)$. Let $N_i(q) = |P_i(q_0, q)|$ be the number of paths of length i from the vertex q_0 to vertex q . We build up N_i by induction on the path length i as follows.

Theorem 2. *For any vertex q and integer $i > 0$, $N_{i+1}(q) = \sum_{e=(p,q) \in E} N_i(p) \cdot W(e)$. If $q = q_0$ then $N_0(q) = 1$, else $N_0(q) = 0$.*

4.4 Path Length

To insure an accurate count of the accepting paths in a DFA, it is important to take into account their length. The fundamental reason for this is that the map from values to representations is one-to-many, since any representation of a value may be arbitrarily extended with leading 0s without changing the value it represents. For example, the DFA in Fig. 3 recognizes at least two different encodings for the values $i = 8$, $j = 5$, $k = 12$, and $R = 2$. Two possible paths are

$$P_4(1, 10) = \begin{smallmatrix} 1000 \\ 0101 \\ 1100 \\ 0010 \end{smallmatrix} \text{ and } P_5(1, 10) = \begin{smallmatrix} 01000 \\ 00101 \\ 00100 \\ 00010 \end{smallmatrix}.$$

Despite the fact that these *two* different encodings are recognized by the DFA, they specify only *one* solution. We want to avoid counting this solution twice, and we do so by counting all solutions identified by accepting paths of the same length (*i.e.*, by encoding the values of all free variables in the same number of bits). For any choice of length L , the number of accepting paths of that length in graph $G(M)$ is equivalent to the number of solutions to the Presburger formula represented by DFA M such that the value of each free variable comprising the solution is in the range $[0, 2^L - 1]$.

In order for counting to be feasible, the number of solutions to the Presburger formula must be finite. The Presburger formulas of interest to us are representable by bounded polytopes, where the number of formula solutions corresponds to the number of integer points in the polytopes. The bounded nature of the polytopes ensures a finite number of solutions.

Recall from Sect. 4.2 that accepting paths encode the binary representations of integer values satisfying the Presburger formula represented by the DFA (LSB to MSB). Each integer value has a unique binary representation with the exception of leading 0s. Therefore, a unique formula solution is represented in the DFA by a single accepting path that may be arbitrarily extended with 0s. Let R be the regular expression representing the set of all strings accepted by a DFA that recognizes a Presburger formula P . In order for P to have a finite set of solutions (in the value domain), the Kleene star operator can appear only in limited positions in R .

Lemma 1. *Let DFA M recognize Presburger formula P , and let regular expression R represent all accepting paths of M . Formula P has a finite number of solutions if and only if R is of the form 0^*S , where S is a regular expression free of the Kleene star operator.*

Lemma 1 gives a property of accepting DFA paths that is critical to our counting algorithm, and it relates the property to the finiteness of Presburger formula solution counts. *Note that this property does not apply to non-accepting paths.* For example, consider the DFA in Fig. 3, which recognizes a Presburger formula with a finite number of solutions. The non-accepting path that goes through nodes 1, 5, and 2 violates the 0^*S pattern, since the self-loop at the beginning of the path arbitrarily extends it with 0s or 1s.

Let V_i be the set of vertices reachable from vertex q_0 via paths of length i . To start, q_0 is the only vertex reachable from itself with path length 0 (*i.e.*, $V_0 = \{q_0\}$). In general, the set of vertices reachable from vertex q_0 with path length i is $V_i = \Delta(V_{i-1})$.

The following theorem establishes that when $V_i = V_{i-1}$, the sets of vertices reachable from vertex q_0 via paths of length $k \geq i$ are identical. In other words, the set of vertices reachable from the starting vertex with paths of a certain length reaches a steady state.

Theorem 3. *Given a DFA M corresponding to a Presburger formula with a finite number of solutions, if $V_i = V_{i-1}$, then $V_k = V_{i-1}$, $\forall k \geq i$.*

Now we want to relate the condition for the set of vertices reaching a steady state to the number of accepting paths. The following theorem shows that when $V_i = V_{i-1}$, the number of paths of length i from q_0 to accepting vertices is the same as the number of such paths of length $i - 1$.

Theorem 4. *Given a DFA M corresponding to a Presburger formula with a finite number of solutions, if $V_i = V_{i-1}$, then $\sum_{q \in F \cap V_i} N_i(q) = \sum_{q \in F \cap V_{i-1}} N_{i-1}(q)$.*

Given Theorems 3 and 4, we know that when the set of vertices reaches a steady state ($V_k = V_{i-1}$, $\forall k \geq i$), the number of accepting paths does as well ($\sum_{q \in F} N_k(q) = \sum_{q \in F} N_{i-1}(q)$, $\forall k \geq i$). Notice that set V_k can contain garbage states, but the paths reaching steady state are only the accepting paths. Our counting algorithm terminates when $V_i = V_{i-1}$ because we can be sure that the number of accepting paths in the DFA has converged.

4.5 The Counting Algorithm

The algorithm shown in Fig. 4 counts the number of accepting paths of length L in a directed, edge-weighted graph $G(M)$. We assume a representation of DFA $M = (S, \Sigma, \delta, q_0, F)$ that includes the following pieces of information.

1. **states**, a list of all states in the DFA;
2. **final**, a flag for each state p indicating if $p \in F$;
3. **to**, for each state p a list of the states q such that there exists a transition from p to q ; and
4. **trans**, a transition table such that an element p, q is the list of alphabet symbols causing transition from state p to state q (note that in general this table is quite sparse).

Algorithm 1. Counting solutions to Presburger formula.

Input: DFA $M = (S, \Sigma, \delta, q_0, F)$ corresponding to Presburger formula.

Output: Path length L , number of solutions to original Presburger formula such that the value of each free variable is in the range $[0, 2^L - 1]$.

Method:

```

1 Construct the graph  $G(M) = (V, E, W)$  from  $M$ .
2  $V_0 \leftarrow \{q_0\}$ 
3  $i \leftarrow 0$ 
4 repeat
5    $i \leftarrow i + 1$ 
6    $V_i \leftarrow \emptyset$ 
7   for all  $q \in V : \exists p \in V_{i-1} \wedge (p, q) \in E$  do
8      $V_i \leftarrow V_i \cup q$ 
9     Calculate  $N_i(q)$  using Theorem 2.
10  enddo
11 until  $V_i = V_{i-1}$ 
12  $L \leftarrow i - 1$ 
13 return  $L, \sum_{q \in F} N_L(q)$ 

```

Fig. 4. Algorithm 1 counts solutions to a Presburger formula.

Notice that we are counting the number of accepting paths in a directed graph *without enumerating each accepting path*. Therefore, the cost of our counting algorithm is sublinear in the number of solutions. The cost of line 1 is $O(|V| + |E|)$. The cost of lines 7–10 is $O(|V|^2)$, at worst. The cost of line 13 is $O(|V|)$. Finally, the worst-case bound on the complexity of the entire algorithm is $O(|V|^3)$. Empirical results in Sect. 5 suggest that the complexity is actually subquadratic in $|V|$. For a more detailed breakdown of Algorithm 1’s complexity, see [29].

5 Examples

Our original experiment considered eight example problems. Because of space constraints, this section gives results of using our counting algorithm on three of those examples⁴ (see [29] for the complete set of results). Examples 1 and 2 come from related work [1,2] and are handled by Clauss [1], while Pugh [9] handles Example 1. Example 3 comes from [3] and is included to show the generality and robustness of our algorithm. We use the DFA-construction algorithms of Bartzis and Bultan [22,23] to construct the DFA representation of the Presburger formula in each example. All running times were collected on a 450MHz Sparc Ultra 60. Note that Pugh and Clauss give no running times for the application of their techniques, so we cannot compare our running times to theirs.

⁴ All three examples require geometric preprocessing to express them as disjoint unions of polyhedra, which prevents us from comparing our counting method with the LattE tool’s [15] implementation of Barvinok’s algorithm (see Sect. 1).

a.

n	# of formula solutions	DFA time (ms)	# of DFA states	count time (ms)	L		n	# of formula solutions	DFA time (ms)	# of DFA states	count time (ms)	L
4	4	64	6	0.041			22	44	91	22	0.107	
5	5	65	7	0.045			23	46	91	19	0.100	
6	6	65	7	0.045			24	48	86	19	0.098	
7	7	67	7	0.044	3		25	50	93	23	0.115	
8	8	65	8	0.050			26	52	86	23	0.114	
9	9	70	9	0.054			27	54	92	21	0.105	
10	10	67	9	0.055			28	56	90	21	0.130	
11	11	71	9	0.052			29	58	92	23	0.114	
12	12	69	9	0.053	4		30	60	91	23	0.115	
13	13	72	10	0.063			31	62	92	18	0.091	
14	14	70	10	0.064			32	64	76	19	0.095	
15	15	72	9	0.051			33	97	104	35	0.255	
16	16	69	10	0.062			34	102	88	35	0.243	6
17	32	85	21	0.110			35	105	109	32	0.227	
18	36	75	22	0.110			36	108	101	32	0.228	
19	38	92	20	0.103			100	700	148	50	0.475	7
20	40	90	20	0.101			500	16,000	212	76	0.826	9
21	42	91	22	0.111			1000	63,000	243	96	1.112	10
							10,000	6,250,000	770	151	2.368	14

μ_a	n	# of formula solutions	DFA time (ms)	# of DFA states	count time (ms)	L
0	100	625	23	19	0.142	10
	1000	62,500	3604	30	0.188	16
21,324	100	626	35	28	0.179	10
	1000	62,501	3526	46	0.365	16

$*m, p$	# of formula solutions	DFA time (ms)	# of DFA states	count time (ms)	L
10	902	3	20	0.152	5
100	611,252	6	48	0.608	8
1000	586,087,502	19	81	1.073	11
10,000	583,608,375,002	205	115	1.975	14

b.

c.

$*n$	$m = m + 1$
10	902
100	611,252
1000	586,087,502
10,000	583,608,375,002

c.

Fig. 5. Results of using our counting algorithm on Examples 1a (a.), 1b (b.), 2(c.), and 3(d.).

We use the following notation in the examples below. The formula P_i pertains to Example i . The notation $(\Sigma : G : x)$ indicates a guarded sum: if G is true, the value of the expression is x , otherwise it is 0. In the tables below, results include the number of solutions to the Presburger formula with n instantiated, the time required for DFA construction (given in milliseconds), the number of states in the DFA representation of the formula, the time required to count the number of accepting paths in the DFA (given in milliseconds), and the length of accepting DFA paths L . Horizontal lines in the tables indicate a change in L .

Example 1. Calculate the number of distinct cache lines touched by the loop nest in Fig. 6.a, which performs Successive Over-Relaxation. Pugh chooses a simple way of mapping array elements to cache lines, suggesting that array element $a[i, j]$ maps to cache line $[(i - 1)/16, j]$. Clauss does the same. We acknowledge that this mapping relation is simplistic, but use it to get our first set of results. Later we extend this example to consider a more interesting mapping relation.

The problem that Pugh and Clauss consider is described by the following Presburger formula,

$$\begin{aligned} P_{1a}(x, y; n) = \exists i, j : 2 \leq i \leq n-1 \wedge 2 \leq j \leq n-1 \wedge ((16x \leq i-1 < 16(x+1) \wedge y = j) \vee \\ (16x \leq (i-1)-1 < 16(x+1) \wedge y = j) \vee (16x \leq (i+1)-1 < 16(x+1) \wedge y = j) \\ \vee (16x \leq i-1 < 16(x+1) \wedge y = j-1) \vee (16x \leq i-1 < 16(x+1) \wedge y = j+1)). \end{aligned}$$

The table in Fig. 5.b gives the results of using our algorithm for 37 values of n . We consider $n = 4$ to 36 because Clauss does in [1]. We consider $n = 500$ because Pugh does in [9]. We also consider three other values of n . Pugh expresses the number of solutions as $(\Sigma : 3 \leq n : n(1 + \lfloor (n-2)/16 \rfloor)) + (\Sigma :$

```

do i = 2, n-1
  do j = 2, n-1
    a[i,j] = (2*a[i,j]+a[i-1,j]
      + a[i+1,j]+a[i,j-1]
      + a[i,j+1])/6
  enddo
enddo

```

a.


```

do i = 0, n
  do j = 0, i+m/2
    do k = 0, i-n+p
      a[i,j,k] = a[i,j,k-1] +
        a[k,j,k]*a[i,k,k]
    enddo
  enddo
enddo

```

b.

Fig. 6. Loop nests for the problems in Examples 1 (a.) and 2 (b.).

$n \bmod 16 = 1 \wedge n \geq 17 : n - 2)$ Clauss expresses the number of solutions as $(\sum : 1 \leq n : \frac{1}{16}n^2 + [\frac{15}{16}, \frac{7}{8}, \frac{13}{16}, \frac{3}{4}, \frac{11}{16}, \frac{5}{8}, \frac{9}{16}, \frac{1}{2}, \frac{7}{16}, \frac{3}{8}, \frac{5}{16}, \frac{1}{4}, \frac{3}{16}, \frac{1}{8}, \frac{1}{16}, 0]n + [-2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])$. We have verified that the solution counts given by our algorithm match those given by both Pugh and Clauss.

Now we extend this example to consider a more interesting mapping relation. An array element $a[i, j]$ maps to cache line $c = \lfloor (\mu_a + \beta((i-1) + n(j-1))/B \rfloor$, where μ_a is the starting address of array a in memory, β is the array element size in bytes, and B is the blocksize of the cache in bytes. Let $\beta = 4$ and $B = 64$. This problem is described by the following first-order logic formula,

$$P_{1b}(c; \mu_a, n) = \exists i, j : 2 \leq i \leq n-1 \wedge 2 \leq j \leq n-1 \wedge (64c \leq \mu_a + 4(i-1+n(j-1)) < 64(c+1) \vee 64c \leq \mu_a + 4((i-1)-1+n(j-1)) < 64(c+1) \vee 64c \leq \mu_a + 4((i+1)-1+n(j-1)) < 64(c+1) \vee 64c \leq \mu_a + 4(i-1+n((j-1)-1)) < 64(c+1) \vee 64c \leq \mu_a + 4(i-1+n((j+1)-1)) < 64(c+1)).$$

Note that formula $P_{1b}(c; \mu_a, n)$ is not a Presburger formula unless the value of n is instantiated, as $n \cdot j$ is not a term of Presburger arithmetic. Speaking of his simple way of mapping array elements to cache lines in [9], Pugh suggests, “We could also assume more general mappings, in which the cache lines can wrap from one row to another and in which we don’t know the alignment of the first element of the array with the cache lines.” Such a mapping is exactly what we have above, and the formulation of the general mapping is not expressible in Presburger arithmetic for symbolic n . Therefore, Pugh would not be able to handle such a mapping without instantiating n as we have done.

We consider a version of formula $P_{1b}(c; \mu_a, n)$ simplified using the Omega Calculator [30,31]. The table in Fig. 5.c gives the results of using our algorithm for $\mu_a \in \{0, 21324\}$ and $n \in \{100, 1000\}$. Note that starting address $\mu_a = 0$ aligns with a cache line boundary and $\mu_a = 21,324$ does not, accounting for the different solution counts. Also note that solution counts differ between formulas P_{1a} and P_{1b} , with counts for the latter formula being no larger than the former. This is as expected, being the result of cache lines wrapping across rows of the array.

Example 2. Calculate the number of flops for the loop nest in Fig. 6.b. More precisely, Clauss calculates the number of iterations in the loop nest. This problem is described by the Presburger formula $P_3(i, j, k; m, n, p) = 0 \leq i \leq n \wedge 0 \leq$

p m	1						2						3						4					
	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)
1	29	3	9	0.069	56	3	10	0.080	90	3	11	0.090	130	3	12	0.096	130	3	12	0.096	130	3	15	0.177
2	32	3	9	0.068	62	3	13	0.168	100	3	12	0.093	145	3	15	0.177	145	3	15	0.176	145	3	15	0.176
3	32	3	9	0.069	62	3	13	0.166	100	3	12	0.094	145	3	15	0.176	145	3	15	0.176	145	3	15	0.176
4	35	3	9	0.071	68	3	12	0.093	110	3	13	0.168	160	3	13	0.096	160	3	13	0.096	160	3	13	0.096
5	35	3	9	0.072	68	3	12	0.091	110	3	13	0.167	160	3	13	0.098	160	3	13	0.098	160	3	13	0.098
6	38	3	9	0.068	74	3	13	0.168	120	3	12	0.091	175	3	15	0.179	175	3	15	0.179	175	3	15	0.179
7	38	3	9	0.070	74	3	13	0.167	120	3	12	0.091	175	3	15	0.181	175	3	15	0.181	175	3	14	0.173
8	41	3	9	0.069	80	3	12	0.095	130	3	13	0.168	190	3	14	0.173	190	3	14	0.173	190	3	14	0.173

p m	5						6						7						8					
	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)
1	175	3	13	0.164	224	2	13	0.098	276	3	15	0.177	330	2	14	0.173	330	2	14	0.173	330	2	18	0.129
2	196	3	14	0.170	252	3	17	0.127	312	3	16	0.181	375	3	18	0.129	375	3	18	0.129	375	3	18	0.129
3	196	3	14	0.171	252	3	17	0.124	312	3	16	0.183	375	3	18	0.130	375	3	18	0.130	375	3	18	0.130
4	217	3	15	0.177	280	3	16	0.185	348	3	17	0.124	420	3	16	0.180	420	3	16	0.180	420	3	16	0.180
5	217	3	15	0.178	280	3	16	0.186	348	3	17	0.125	420	3	16	0.178	420	3	16	0.178	420	3	18	0.127
6	238	3	14	0.172	308	3	17	0.129	384	3	15	0.278	465	3	18	0.127	465	3	18	0.127	465	3	18	0.127
7	238	3	14	0.171	308	3	17	0.127	384	3	15	0.181	465	3	18	0.132	465	3	18	0.132	465	3	17	0.125
8	259	3	15	0.182	336	3	15	0.177	420	3	17	0.128	510	3	17	0.125	510	3	17	0.125	510	3	17	0.125

Fig. 7. Results of using our counting algorithm on Presburger formula $P_2(i, j, k; m, n, p)$ for $m = 1$ to 8, $n = 9$, and $p = 1$ to 8. In all cases, the path length L is 4.

$j \leq i + m/2 \wedge 0 \leq k \leq i - n + p$. We consider $m = 1$ to 8, $n = 9$, and $p = 1$ to 8 because Clauss does in [1], and show those results in Fig. 7. In the table in Fig. 5.d, we consider larger values of m , n , and p . Clauss expresses the number of solutions as $(\Sigma : n > p : \frac{1}{2}np^2 + \frac{3}{2}np + n + \frac{1}{4}mp^2 + \frac{3}{4}mp + \frac{1}{2}m - \frac{1}{6}p^3 + [-\frac{1}{4}, 0]mp^2 + [\frac{5}{12}, \frac{7}{6}]mp + [\frac{1}{2}, 1]_m)$. We have verified that our solution counts match those of Clauss.

Example 3. Consider Presburger formula $P_3(j_1, j_2, s, d; \mu_X)$ in Fig. 8. This formula represents the interior misses incurred by array X due to interference from array A during a matrix-vector multiplication. See [3] for a more detailed description of the matrix-vector multiply loop nest and the execution parameters. In the formula, μ_X is a symbolic constant and it represents the starting address of array X in memory. The table in Fig. 5.e gives the results of using our algorithm for three values of μ_X . We have verified that our solution counts match the interior miss counts reported by a (specially-written) cache simulator. Example 3 demonstrates our claim that our algorithm can handle non-smooth formulas. The other three examples are quite continuous, but this formula is very “spiky” for various values of μ_X (as Fig. 2 in Fricker *et al.* [12] shows).

Observations. The following are five noteworthy observations on the results.

1. The number of DFA states does not grow significantly as the values of the size constants increase, showing good scalability.
2. For all 160 DFAs considered in the original experiment (see [29]), the curve $y = 0.0025x^{1.4}$ fits the plot of the time required to count all accepting paths against the number of DFA states, illustrating a subquadratic relationship between the number of states and counting time. Notice that counting time is markedly sublinear in the number of solutions.
3. The running times show that the counting algorithm is far less expensive than the DFA-construction algorithms of Bartzis and Bultan [22,23]. However, both are clearly quite fast.

$$\begin{aligned}
(j_2 = 0 \wedge (\exists \alpha : 1 \leq j_1 \leq 99 \wedge 0 \leq s \leq 255 \wedge \alpha < d \wedge 32s + 8192d \leq \mu_X \wedge 800j_1 + 8192d \leq 792 + \mu_X + 8192\alpha \\
\wedge \mu_X \leq 31 + 32s + 8192d \wedge s + 256\alpha < 25j_1)) \vee (\exists \alpha : 1 \leq j_1 \leq 99 \wedge 0 \leq s \leq 255 \wedge 1 \leq j_2 \wedge \\
925 + 100j_1 + j_2 \leq 1024d + 4s \wedge 8192d + 32s \leq \mu_X + 8j_2 \wedge \mu_X - 8j_2 \leq 7 + 8192d + 32s \wedge \\
100j_1 + j_2 \leq 99 + 4s + 1024\alpha \wedge s + 256\alpha < 25j_1) \vee (\exists \alpha : 0 \leq j_1 \leq 99 \wedge 0 \leq s \leq 255 \wedge j_2 \leq 99 \wedge \\
25j_1 \leq s + 256\alpha \wedge 8192d + 32s \leq \mu_X + 8j_2 \wedge \mu_X + 8j_2 \leq 7 + 8192d + 32s \wedge 4s + 1024\alpha < 100j_1 + j_2 \\
\wedge 256 + 25j_1 \leq 256d - s) \vee (\exists \alpha : 0 \leq j_1 \leq 99 \wedge 0 \leq j_2 \leq 99 \wedge 0 \leq s \leq 255 \wedge 8192d + 32s \leq \mu_X + 8j_2 \wedge \\
\mu_X + 8j_2 \leq 31 + 8192d + 32s \wedge 1021 + 100j_1 + j_2 \leq 1024d + 4s \wedge 100j_1 + j_2 \leq 3 + 4s + 1024\alpha \wedge \\
4s + 1024\alpha \leq 100j_1 + j_2) \vee (j_2 = 0 \wedge (\exists \alpha : 1 \leq j_1 \leq 99 \wedge 0 \leq s \leq 255 \wedge 257 + s + 256d \leq 25j_1 \wedge \\
32s + 8192d \leq \mu_X \wedge 800j_1 + 8192d \leq 792 + \mu_X + 8192\alpha \wedge \mu_X \leq 31 - 32s + 8192d \wedge s + 256\alpha < 25j_1)) \\
\vee (\exists \alpha : 1 \leq j_1 \leq 99 \wedge 0 \leq s \leq 255 \wedge 1 \leq j_2 \wedge 257 + 256d + s \leq 25j_1 \wedge 8192d + 32s \leq \mu_X + 8j_2 \wedge \\
\mu_X + 8j_2 \leq 7 + 8192d + 32s \wedge 100j_1 + j_2 \leq 99 + 4s + 1024\alpha \wedge s + 256\alpha < 25j_1) \vee (\exists \alpha : 0 \leq j_1 \leq 99 \wedge \\
0 \leq s \leq 255 \wedge j_2 \leq 99 \wedge 25j_1 \leq s + 256\alpha \wedge 8192d + 32s \leq \mu_X + 8j_2 \wedge \mu_X + 8j_2 \leq 7 + 8192d + 32s \wedge \\
4s + 1024\alpha < 100j_1 - j_2 \wedge 1025 + 1024d + 4s \leq 100j_1 + j_2) \vee (\exists \alpha : 0 \leq j_1 \leq 99 \wedge 0 \leq j_2 \leq 99 \wedge \\
0 \leq s \leq 255 \wedge 8192d + 32s \leq \mu_X + 8j_2 \wedge \mu_X + 8j_2 \leq 31 + 8192d + 32s \wedge 1024 - 1024d + 4s \leq 100j_1 + j_2 \\
\wedge 100j_1 + j_2 \leq 3 + 4s + 1024\alpha \wedge 4s + 1024\alpha \leq 100j_1 + j_2)
\end{aligned}$$

Fig. 8. Presburger formula $P_3(j_1, j_2, s, d; \mu_X)$ describing interior misses on array X due to interference from array A during matrix-vector multiply.

4. Only formulas $P_{1b}(c; \mu_a, n)$ and $P_3(j_1, j_2, s, d; \mu_X)$ require simplification before our counting method is applied. For formula P_{1b} , the total amount of time to simplify all instantiations is less than 0.5 seconds. For formula P_3 , the total amount of time to simplify all instantiations is less than 0.3 seconds. The Omega Calculator [30,31] is used for all simplifications.
5. The examples presented here contain large values (order 10,000). The fast running times demonstrate that our counting method can handle such values.

6 Conclusions

We have presented an automata-theoretic algorithm for counting integer solutions to selected free variables of a Presburger formula. Demonstration of our counting algorithm on three examples shows that it is robust, universal, fast, and scalable. The time required to count all accepting paths scales well with the number of DFA states and solutions.

The polyhedral methods of Pugh and Clauss and our automata-theoretic method have complementary strengths. While the formulas in Figs. 1 and 2.b would produce the same DFA in the end, the DFA construction time for the formula of Fig. 1 would be prohibitively large. We therefore routinely simplify complicated Presburger formulas such as the one in Fig. 1, P_{1b} , and P_3 as much as possible using the Omega Calculator before constructing the DFAs and counting solutions. On the other hand, by instantiating the symbolic parameters, our counting algorithm can handle non-smooth formulas that are too difficult to handle symbolically using polyhedral methods. Therefore, in cases where the symbolic methods of Pugh and Clauss can be used simply and quickly, they should be. For cases which are difficult or impossible for the methods of Pugh and Clauss to handle, our counting algorithm is a nice alternative. Our algorithm could also be effectively used as a subroutine in Clauss' method to count the number of solutions for the necessary number of small-sized polyhedra.

References

1. Clauss, P.: Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In: Proceedings of the 1996 ACM International Conference on Supercomputing. (1996) 278–285
2. Ferrante, J., Sarkar, V., Thrash, W.: On estimating and enhancing cache effectiveness. In Banerjee, U., et al., eds.: Proceedings of the Fourth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Volume 589 of Lecture Notes in Computer Science., Springer (1991) 328–343
3. Chatterjee, S., Parker, E., Hanlon, P.J., Lebeck, A.R.: Exact analysis of the cache behavior of nested loops. In: Proceedings of 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation. (2001) 286–297
4. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: A compiler framework for analyzing and tuning memory behavior. ACM Transactions on Programming Languages and Systems **21** (1999) 703–746
5. Haghigat, M., Polychronopoulos, C.: A basis for parallelization, optimization and scheduling of programs. Technical Report 1317, CSRD, University of Illinois (1993)
6. Haghigat, M., Polychronopoulos, C.: Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. In Banerjee, U., et al., eds.: Proceedings of the Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Volume 768 of Lecture Notes in Computer Science., Springer (1993) 567–585
7. Tawbi, N.: Estimation of nested loop execution time by integer arithmetics in convex polyhedra. In: Proceedings of the Eighth International Parallel Processing Symposium. (1994) 217–221
8. Tawbi, N., Feautrier, P.: Processor allocation and loop scheduling on multiprocessor computers. In: Proceedings of the 1992 ACM International Conference on Supercomputing. (1992) 63–71
9. Pugh, W.: Counting solutions to Presburger formulas: How and why. In: Proceedings of 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation. (1994) 121–134
10. Ehrhart, E.: Polynômes arithmétiques et Méthode des Polyédres en Combinatoire. Volume 35 of International Series of Numerical Mathematics. Birkhäuser Verlag, Basel/Stuttgart (1977)
11. Loechner, V.: PolyLib: A Library for Manipulating Parameterized Polyhedra. (1999)
12. Fricker, C., Temam, O., Jalby, W.: Influence of cross-interference on blocked loops: A case study with matrix-vector multiply. ACM Transactions on Programming Languages and Systems **17** (1995) 561–575
13. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Mathematics of Operations Research **19** (1994) 769–779
14. Dyer, M., Kannan, R.: On Barvinok’s algorithm for counting lattice points in fixed dimension. Mathematics of Operations Research **22** (1997) 545–549
15. Loera, J.A.D., Hemmecke, R., Tauzer, J., Yoshida, R.: Effective lattice point counting in rational convex polytopes.
<http://www.math.ucdavis.edu/~latte/pdf/lattE.pdf> (2003)

16. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves. (1929) 92–101
17. Oppen, D.C.: A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. Journal of Computer and System Sciences **16** (1978) 323–332
18. Weispfenning, V.: Complexity and uniformity of elimination in Presburger arithmetic. In: Proceedings of the 1997 ACM SIGSAM International Symposium on Symbolic and Algebraic Computation. (1997) 48–53
19. Büchi, J.R.: Weak second-order arithmetic and finite automata. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik **6** (1960) 66–92
20. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In Nagel, E., et al., eds.: Proceedings of the International Congress on Logic, Methodology and Philosophy of Science 1960, Stanford University Press (1962) 1–11
21. Boudet, A., Comon", H.: Diophantine equations, Presburger arithmetic and finite automata. In Kirchner, H., ed.: Proceedings of 21st International Colloquium on Trees in Algebra and Programming. Volume 1059 of Lecture Notes in Computer Science., Springer (1996) 30–43
22. Bartzis, C., Bultan, T.: Efficient symbolic representations for arithmetic constraints in verification. Technical Report 2002-16, Computer Science Department, University of California, Santa Barbara (2002)
23. Bartzis, C., Bultan, T.: Automata-based representations for arithmetic constraints in automated verification. In: Proceedings of the 7th International Conference on Implementation and Application of Automata. (2002) 282–288
24. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company (1979)
25. Boigelot, B., Wolper, P.: Representing arithmetic constraints with finite automata: An overview. In J.Stuckey, P., ed.: Proceedings of the 18th International Conference on Logic Programming. Volume 2401 of Lecture Notes in Computer Science., Springer (2002) 1–19
26. Wolper, P., Boigelot, B.: An automata-theoretic approach to Presburger arithmetic constraints (extended abstract). In Mycroft, A., ed.: Proceedings of the Second International Static Analysis Symposium. Volume 983 of Lecture Notes in Computer Science., Springer (1995) 21–32
27. Wolper, P., Boigelot, B.: On the construction of automata from linear arithmetic constraints. In Graf, S., Schwartzbach, M.I., eds.: Proceedings of the Sixth International Conference on Tools and Algorithms for Construction and Analysis of Systems. Volume 1785 of Lecture Notes in Computer Science., Springer (2000) 1–19
28. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. (2001)
<http://www.brics.dk/mona>.
29. Parker, E., Chatterjee, S.: An automata-theoretic algorithm for counting solutions to Presburger formulas. Technical Report TR04-001, Department of Computer Science, University of North Carolina at Chapel Hill (2004)
30. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega Calculator and Library, Version 1.1.0. (1996)
<http://www.cs.umd.edu/projects/omega>.
31. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega Library Version 1.1.0 Interface Guide. (1996)
<http://www.cs.umd.edu/projects/omega>.

A Symbolic Approach to Bernstein Expansion for Program Analysis and Optimization

Philippe Clauss¹ and Irina Tchoupaeva²

¹ INRIA Rocquencourt, A3 Project
78153 Le Chesnay Cedex - France
clauss@icps.u-strasbg.fr

² ICPS/LSIIT, Université Louis Pasteur, Strasbourg
Pôle API, Bd Sébastien Brant
67400 Illkirch - France
irina@icps.u-strasbg.fr

Abstract. Several mathematical frameworks for static analysis of programs have been developed in the past decades. Although these tools are quite useful, they have still many limitations. In particular, integer multivariate polynomials arise in many situations while analyzing programs, and analysis systems are unable to handle such expressions. Although some dedicated methods have already been proposed, they only handle some subsets of such expressions. This paper presents an original and general approach to Bernstein expansion which is symbolic. Bernstein expansion allows bounding the range of a multivariate polynomial over a box and is generally more accurate than classic interval methods.

1 Introduction

Static program analysis has received a lot of attention in the past decades as mathematical frameworks for modeling programs have been developed. For example, the polytope model allows modeling nested loops as polyhedra whose contained integer points are associated to iterations. It provides mathematical tools for quantitative analysis and optimizing transformations. However this model is limited to loops whose loop bounds are affine functions over the enclosing loop indices, and array references are affine functions over the loop indices.

Linearity of the considered functions is one of the main limitations in formal analysis. Many behavior modeling issues where interactions between hardware and software are considered, as cache behavior of programs, involve nonlinear expressions. Compilers implementing advanced optimization transformations are also limited by this fact. In particular, multivariate polynomial functions arise in many situations resulting from linearized subscripts in array reference functions, or from induction variable recognition, and compilers fail in handling such expressions.

In this paper, we propose an extension of the theory of Bernstein expansion [3,4] to handle parameterized multivariate polynomial expressions. Bernstein expansion allows bounding the range of a multivariate polynomial considered

over a box [2,11]. Numerical applications of this theory have been proposed to the resolution of system of strict polynomial inequalities [12,13]. It has been shown that Bernstein expansion is generally more accurate than classic interval methods [14]. Moreover, Stahl has shown in [17] that for *sufficiently small* boxes, the exact range is obtained.

In this paper, we generalize Bernstein expansion by considering parameterized multivariate polynomials for which ranges are bound by polynomials over the parameters. Sufficient conditions over the parameters for strict parameterized inequalities solutions can therefore be constructed.

Bernstein polynomials are particular polynomials that form a basis for the space of polynomials. Hence any polynomial can be expressed into this basis through coefficients, the Bernstein coefficients, that have interesting properties and that can be computed through a direct formula. Due to the Bernstein convex hull property [10], the value of the polynomial is then bounded by the values of the minimum and maximum Bernstein coefficients. The direct formula allows symbolic computation of these Bernstein coefficients giving a supplementary interest to the use of this theory. Another main interesting consequence is that the involved computations have quite low complexity. Moreover an appropriate instrumentation of this model allows automatic and inexpensive resolutions of complex program analysis issues.

This paper is organized as follows. Following a short and motivating application example in section 2, some notations are introduced and the handled polynomials are defined in section 3. In section 4, basic notions about Bernstein expansion are recalled as well as the general principles of its numerical application. Our symbolic approach is presented in section 5 where handled parameterized multivariate polynomials inequalities over parameterized boxes are considered. Several application examples are described in section 6: dependence testing between references with linearized subscripts, dead code elimination of conditional statements and computation of the minimum memory amount needed for array contraction. In section 7, we point out some current limits of our framework and discuss about some further developments. In this section, we also compare our approach to related works. Conclusions are finally given in section 8.

2 A Motivating Example

Consider the following loop, where “...” symbolizes some instructions having no effect on the considered ones:

```
q=alpha-25 ;
for (i=0 ; i<11 ; i++) {
    ...
    a[q]+=a[i+alpha+1] ;
    ...
    q-=i ;
    ...
    q+=9 ;
```

```

...
q-=i ;
...
}
```

In order to determine whether this loop can be parallelized, it has to be determined if array reference $a[q]$ overlaps reference $a[i+\alpha+1]$. For this, some induction variable recognition is first achieved [16] in order to get an expression of q depending on i :

```

for (i=0 ; i<11 ; i++) {
...
a[-i*i+10*i+alpha-25]+=a[i+alpha+1] ;
... }
```

By applying conventional value range analysis on both polynomials $-i^2 + 10i + \alpha - 25$ and $i + \alpha + 1$, we get the following results:

$$\begin{aligned} -[0, 10]^2 + 10[0, 10] + \alpha - 25 &= [\alpha - 125, \alpha + 75] \\ [0, 10] + \alpha + 1 &= [\alpha + 1, \alpha + 11] \end{aligned}$$

It yields the conclusion that the loop cannot be parallelized, since the value range of $i + \alpha + 1$ is entirely included in the value range of $-i^2 + 10i + \alpha - 25$. But since the form in which the polynomial is expressed can affect the result, we now use the Horner form of the polynomial $-i^2 + 10i + \alpha - 25 = \alpha - 25 + (10 - i)i$ which gives the more accurate bound $[\alpha - 25, \alpha + 75]$, yielding anyway to the same conclusion as before.

By applying our symbolic Bernstein expansion approach presented in this paper, we compute the Bernstein coefficients of $-i^2 + 10i + \alpha - 25$: $b_0 = \alpha$, $b_1 = \alpha - 25$ and $b_2 = \alpha$. Since the value is bounded by the minimum and maximum coefficients, it yields the *exact* value range $[\alpha - 25, \alpha]$. Moreover, it yields the conclusion that ranges of both references do not overlap at all and hence that the loop *can be parallelized*.

Notice that other more accurate methods recently proposed [5,6,9] also would have failed in this case, since the considered polynomial $-i^2 + 10i + \alpha - 25$ is not monotonous over the interval $[0, 10]$.

3 Considered Polynomials and Notations

Let $x = (x_1, \dots, x_{l_x})$ denote a l_x -dimensional vector of rational variables and $u = (u_1, \dots, u_{l_u})$ a l_u -dimensional vector of rational parameters.

$$\begin{aligned} \mathbb{Q}^{l_x} &= \{\bar{x} = (x_1, \dots, x_{l_x}) \mid x_i \in \mathbb{Q}, 1 \leq i \leq l_x\} \\ \mathbb{Q}^{l_u} &= \{\bar{u} = (u_1, \dots, u_{l_u}) \mid u_i \in \mathbb{Q}, 1 \leq i \leq l_u\} \end{aligned}$$

Any term $x_1^{i_1} x_2^{i_2} \dots x_{l_x}^{i_{l_x}}$ is noted \bar{x}^{I_x} , where $I_x = (i_1, \dots, i_{l_x})$ is a vector of l_x positive integer values. In the same way, \bar{u}^{I_u} denotes any term $u_1^{i_1} \dots u_{l_u}^{i_{l_u}}$ with $I_u = (i_1, \dots, i_{l_u})$. I_x , respectively I_u , is called the *multi-index* of variable x ,

respectively u . The null vector $(0, \dots, 0)$ is noted $\bar{0}$. In the following, an inequality $I = (i_1, \dots, i_l) \leq J = (j_1, \dots, j_l)$ means that for each $0 \leq k \leq l$ we have $i_k \leq j_k$.

Let $\mathbb{Q}[\bar{u}]$ be the set of polynomials defined by:

$$\mathbb{Q}[\bar{u}] = \{p(\bar{u}) \mid p(\bar{u}) = \sum_{\bar{0} \leq I_u \leq N_u} a_{I_u} \cdot \bar{u}^{I_u}\}$$

where coefficients $a_{I_u} \in \mathbb{Q}$ and $N_u = (n_{u,1}, \dots, n_{u,l_u})$ is called the *maximal multi-degree* of u -variables, *i.e.*, the vector of the greatest degrees of the u -variables.

In this work, we consider polynomials over x -variables with coefficients in $\mathbb{Q}[\bar{u}]$ and with maximal multi-degree $N_x = (n_{x,1}, \dots, n_{x,l_x})$:

$$Q[\bar{u}][\bar{x}] = \{p(\bar{x}, \bar{u}) \mid p(\bar{x}, \bar{u}) = \sum_{\bar{0} \leq I_x \leq N_x} p_{I_x}(\bar{u}) \cdot \bar{x}^{I_x}, p_{I_x}(\bar{u}) \in \mathbb{Q}[\bar{u}]\}.$$

Any polynomial $p(\bar{x}, \bar{u})$ is considered over a box $D_x \times \mathbb{Q}^{l_u}$ where $D_x \subset \mathbb{Q}^{l_x}$. Considered boxes $D_x = [a_1, b_1] \times \dots \times [a_l, b_l]$ are such that the a_i 's or the b_i 's can be polynomials over parameters or variables and such that the knowledge of the q first variables values determines the parametric values of a_{q+1} and b_{q+1} , $q < l$:

$$D_x = [a_2 = pa_2(x_1, \bar{u}), b_2 = pb_2(x_1, \bar{u})] \times [a_3 = pa_3(x_1, x_2, \bar{u}), b_3 = pb_3(x_1, x_2, \bar{u})] \times \dots \times [a_l = pa_l(x_1, x_2, \dots, x_{l-1}, \bar{u}), b_l = pb_l(x_1, x_2, \dots, x_{l-1}, \bar{u})]$$

where the pa_i 's and pb_i 's are polynomials over some variables and parameters. Our aim is to find integer solutions to the inequality:

$$p(\bar{x}, \bar{u}) > 0, \quad \text{where } (\bar{x}, \bar{u}) \in D_x \times \mathbb{Q}^{l_u} \cap \mathbb{Z}^{l_x + l_u}$$

Solutions are expressed as sets D_0 and D_1 , where $D = (D_x \times \mathbb{Q}^{l_u}) = (D_0 \sqcup D_1 \sqcup D_\epsilon)$ such that for all $(\bar{x}, \bar{u}) \in D_0 \cap \mathbb{Z}^{l_x + l_u}$ $p(\bar{x}, \bar{u}) > 0$, for all $(\bar{x}, \bar{u}) \in D_1 \cap \mathbb{Z}^{l_x + l_u}$ $p(\bar{x}, \bar{u}) \leq 0$, and D_ϵ denotes the set where the sign of $p(\bar{x}, \bar{u})$ is indeterminate.

In this paper we generalize the numerical Bernstein method by applying it to parameterized polynomials of the set $Q[\bar{u}][\bar{x}]$. We first recall the Bernstein theory and the numerical method.

4 Numerical Bernstein Method

Let

$$C_N^K = C_{n_1}^{k_1} \cdot \dots \cdot C_{n_l}^{k_l}, \text{ where } \bar{0} \leq K = (k_1, \dots, k_l) \leq N = (n_1, \dots, n_l), (K, N) \in \mathbb{N}_0^l,$$

$$\text{and } C_n^k = \frac{n!}{k!(n-k)!}, 0 \leq k \leq n, (k, n) \in \mathbb{N}_0$$

$$I - J = (i_1 - j_1, \dots, i_l - j_l), \text{ where } I = (i_1, \dots, i_l), J = (j_1, \dots, j_l), I, J \in \mathbb{N}_0^l \\ \alpha^I = \alpha_1^{i_1} \cdot \dots \cdot \alpha_l^{i_l}, \text{ where } \alpha = (\alpha_1, \dots, \alpha_l) \in \mathbb{R}^l$$

For polynomials with constant coefficients:

$$p(\bar{x}) = \sum_{\bar{0} \leq I \leq N} a_I \cdot \bar{x}^I \in \mathbb{R}[\bar{x}], \quad \bar{x} \in D \subset \mathbb{R}^l, a_I \in \mathbb{R}$$

and with maximal multi-degree N , Bernstein polynomials and Bernstein coefficients are defined in the following way.

In the univariate case, the i -th Bernstein polynomial of degree n on the unit interval $[0, 1]$ is defined as:

$$b_{n,i}^{[0,1]}(x) = C_n^i \cdot x^i (1-x)^{n-i}, \quad 0 \leq i \leq n, x \in \mathbb{R}$$

In the multivariate case, the I -th Bernstein polynomial on the unit box $[0, 1]^l$, where $I = (i_1, \dots, i_l)$ and $N = (n_1, \dots, n_l)$ is defined by:

$$B_{N,I}^{[0,1]^l}(\bar{x}) = b_{n_1,i_1}^{[0,1]}(x_1) \cdot \dots \cdot b_{n_l,i_l}^{[0,1]}(x_l), \quad \bar{0} \leq I \leq N, \bar{x} = (x_1, \dots, x_l) \in \mathbb{R}^l$$

Bernstein polynomials are defined on the whole l -dimensional space \mathbb{R}^l and define a basis for polynomials $p(\bar{x})$:

$$p(\bar{x}) = \sum_{\bar{0} \leq I \leq N} b_I^{[0,1]^l} B_{N,I}^{[0,1]^l}(\bar{x}) = \sum_{\bar{0} \leq I \leq N} a_I \cdot \bar{x}^I,$$

where coefficients $b_I^{[0,1]^l}$ are the Bernstein coefficients of the polynomial $p(\bar{x})$ on the unit box $[0, 1]^l$. They are determined by:

$$b_I^{[0,1]^l} = \sum_{\bar{0} \leq J \leq I} \frac{C_I^J}{C_N^J} a_J, \quad \bar{0} \leq I \leq N.$$

Theorem. Let $p(\bar{x})$ be a polynomial of multi-degree N on the unit box $[0, 1]^l$. For all $\bar{x} \in [0, 1]^l$:

$$\min_{\bar{0} \leq I \leq N} b_I^{[0,1]^l} \leq p(\bar{x}) \leq \max_{\bar{0} \leq I \leq N} b_I^{[0,1]^l}$$

Polynomials on any boxes $D \in \mathbb{R}^l$ of any sizes can also be considered since without loss of generality, any nonempty box can be mapped linearly onto the unit box $[0, 1]^l$. Let ϕ be the linear transformation of the box D onto the unit box $[0, 1]^l$. As a polynomial $p(\bar{x})$ is considered on a box D , then the corresponding polynomial on the unit box is $\tilde{p}(\bar{y}) = p(\phi^{-1}(\bar{y}))$, $\bar{y} \in [0, 1]^l$, and the Bernstein coefficients of the polynomial p on the box D are equal to the Bernstein coefficients of the polynomial \tilde{p} on the unit box $[0, 1]^l$: $b_I^D = b_I^{[0,1]^l}$.

Transformation ϕ is defined by:

$$\phi : \quad \bar{x} = (x_1, \dots, x_l) \rightarrow \bar{y} = (y_1, \dots, y_l) \quad \text{where } y_i = \frac{x_i - a_i}{b_i - a_i}, 0 \leq i \leq l \\ [a_1, b_1] \times \dots \times [a_l, b_l] \rightarrow [0, 1]^l, \quad a_i, b_i \in \mathbb{R}$$

Hence the inverse transformation is:

$$\begin{aligned} \phi^{-1} : \bar{y} = (y_1, \dots, y_l) &\rightarrow \bar{x} = (x_1, \dots, x_l) \\ [0, 1]^l &\rightarrow [a_1, b_1] \times \cdots \times [a_l, b_l], \quad a_i, b_i \in \mathbb{R} \end{aligned}$$

where $x_i = \alpha_i \cdot y_i + \beta_i, \alpha_i = (b_i - a_i), \beta_i = a_i, 0 \leq i \leq l, 0 \leq i \leq l$

And $\tilde{p}(\bar{y})$ is such that:

$$\tilde{p}(\bar{y}) = \sum_{\bar{0} \leq P \leq N} \tilde{a}_P \bar{y}^P \quad \text{where } \tilde{a}_P = \alpha^P \cdot \sum_{P \leq K \leq N} a_K \cdot C_K^P \cdot \beta^{K-P}$$

Note that the maximal multi-degree N is the same for $p(\bar{x})$ and $\tilde{p}(\bar{y})$.

To apply the Bernstein method for polynomials over an arbitrary box D , the mapping ϕ is constructed and the Bernstein method is applied to the assigned polynomial $\tilde{p}(\bar{y})$ over the unit box $[0, 1]^l$.

Using the theorem, an algorithm to find solutions to the inequality $p(\bar{x}) > 0, x \in [0, 1]^l$, can be constructed. Using properties of the Bernstein coefficients of the polynomial on the box D and since $b_I^D = b_I^{[0,1]^l}$ for all $\bar{0} \leq I \leq N$, we have:

- if $\min_{\bar{0} \leq I \leq N} b_I^D > 0$, then for all $x \in D, p(x) > 0$
- if $\max_{\bar{0} \leq I \leq N} b_I^D \leq 0$, then for all $x \in D, p(x) \leq 0$

The complete algorithm can be found in [12,13].

5 Symbolic Approach

Our extension considers polynomials $p(\bar{x}, \bar{u})$ with parametric coefficients over boxes whose bounds can also be parametric. Moreover these bounds can also depend on variables.

5.1 Parameterized Multivariate Polynomials

The transformation of any polynomial:

$$p(\bar{x}, \bar{u}) = \sum_{\bar{0} \leq I_x \leq N_x} p_{I_x}(\bar{u}) \cdot \bar{x}^{I_x}, \quad \bar{x} \in D_x, \quad \bar{u} \in \mathbb{Q}^{l_u}$$

into the polynomial:

$$\tilde{p}(\bar{y}, \bar{u}) = \sum_{\bar{0} \leq I_x \leq N_x} \tilde{p}_{I_x}(\bar{u}) \cdot \bar{y}^{I_x}, \quad \bar{y} \in [0, 1]^{l_x}, \quad \bar{u} \in \mathbb{Q}^{l_u}$$

can also be found in the case of parameterized coefficients $p_{I_x}(\bar{u})$ since the formulas are the same:

$$\tilde{p}_{I_x}(\bar{u}) = \alpha^{I_x} \cdot \sum_{I_x \leq K_x \leq N_x} p_{K_x}(\bar{u}) \cdot C_{K_x}^{I_x} \cdot \beta^{K_x - I_x}$$

where like above $D_x = [a_1, b_1] \times \cdots \times [a_{l_x}, b_{l_x}]$, $\alpha_i = (b_i - a_i)$, $\beta_i = a_i$, $0 \leq i \leq l_x$. Hence parameterized Bernstein coefficients can be computed by:

$$b_{I_x}^{[0,1]^l} = \sum_{\bar{0} \leq J \leq I_x} \frac{C_{I_x}^J}{C_{N_x}^J} \tilde{p}_J(\bar{u}), \quad \bar{0} \leq I_x \leq N_x.$$

and the inequality $p(\bar{x}, \bar{u}) > 0$, $\bar{x} \in D_x$, $\bar{u} \in \mathbb{Q}^{l_u}$ holds if $\min_{\bar{0} \leq I_x \leq N_x} b_{I_x}^{D_x} > 0$. But since the Bernstein coefficients are polynomials over the parameters \bar{u} , the minimum of these coefficients is generally difficult to determine. Verifying that all the coefficients are strictly positive is equivalent and the following system is considered: $b_{I_x}^{D_x} > 0$, for all $\bar{0} \leq I_x \leq N_x$.

Moreover, if coefficients $p_{I_x}(\bar{u})$ are linear functions over the parameters \bar{u} , the Bernstein coefficients of the polynomial p on the box D_x are also linear functions of the parameters and the latter system is a system of linear inequalities. Such a system is handled by several tools manipulating linear functions like for example the Polyhedral library *PolyLib* [1].

Example 1. Consider the polynomial $p(\bar{x}, \bar{u}) = (N + M)x_1x_2 + Nx_1 + Mx_2$ with $\bar{x} = (x_1, x_2)$, $\bar{u} = (N, M)$ and $\bar{x} \in D_x = [0, 100] \times [0, 100]$. Hence multi-degree $N_x = (1, 1)$. Box D_x is mapped onto the unit box $[0, 1]^2$ by transformation ϕ defined by $\phi(\bar{x}) = \bar{y} = (y_1 = x_1/100, y_2 = x_2/100)$. Hence polynomial $\tilde{p}(\bar{y}, \bar{u}) = (10000N + 10000M)y_1y_2 + 100Ny_1 + 100My_2$ with $\bar{y} \in [0, 1]^2$. Bernstein coefficients are computed for all $(0, 0) \leq I_x \leq (1, 1)$:

- for $I_x = (0, 0)$: $b_{(0,0)}^{D_x} = \frac{C_0^0 C_0^0}{C_1^0 C_1^0} 0 = 0$;
- for $I_x = (0, 1)$: $b_{(0,1)}^{D_x} = \frac{C_0^0 C_0^1}{C_1^0 C_1^1} 0 + \frac{C_0^0 C_1^1}{C_1^0 C_1^1} 100M = 100M$;
- for $I_x = (1, 0)$: $b_{(1,0)}^{D_x} = \frac{C_0^1 C_0^0}{C_1^1 C_1^0} 0 + \frac{C_1^1 C_0^0}{C_1^1 C_1^0} 100N = 100N$;
- for $I_x = (1, 1)$: $b_{(0,1)}^{D_x} = \frac{C_0^0 C_1^0}{C_1^0 C_1^1} 0 + \frac{C_0^0 C_1^1}{C_1^0 C_1^1} 100M + \frac{C_1^1 C_0^0}{C_1^1 C_1^0} 100N + \frac{C_1^1 C_1^0}{C_1^1 C_1^1} (10000N + 10000M) = 10100N + 10100M$;

Inequality $p(\bar{x}, \bar{u}) \geq 0$ holds when all these Bernstein coefficients are positive. In this example, it can be easily deduced that it holds for $N \geq 0$ and $M \geq 0$.

5.2 Parameterized Boxes

Since D_x has bounds that can depend on parameters or variables, transformation ϕ to map D_x onto the unit box is no more linear and is defined from variables and parameters: $y_i = \frac{x_i - a_i}{b_i - a_i}$ where a_i and b_i can be polynomials over variables and parameters. Hence, ϕ is valid if and only if $b_i - a_i \neq 0$, which means that $a_i < b_i$. So a valid mapping on the unit box can only be defined for D_x where parameters and variables values such that $b_i - a_i = 0$ have been excluded. We note this reduced set D_x^* which is generally a union of disjoint boxes:

$$D_x^* = D_x - \{\bar{x} \in D_x \mid \exists 1 \leq i \leq l, x_i = \text{Root_of}(b_i - a_i)\}$$

When the a_i 's and b_i 's are affine functions of variables and parameters, the roots of $(b_i - a_i)$ are easily computed. Otherwise, roots can also be easily computed for univariate polynomials of degree at most 4. Transformation ϕ is then defined and applied on D_x^* in the following way.

At first, the polynomial $\tilde{p}(\bar{y}, \bar{u})$ is computed as described above. Then an additional transformation consisting in propagating the new variables in $\tilde{p}(\bar{y}, \bar{u})$ is necessary. Finally, the Bernstein coefficients are computed from this polynomial on $[0, 1]^l$. A specific value-per-value analysis is finally done for all roots of the $(a_i - b_i)$'s polynomials.

Example 2. Consider the polynomial $p(i, j) = (i^2 + i)/2 + j$ defined on the box $D_x = [0, N] \times [0, i]$. Transformation ϕ is such that $\phi(i, j) = (i' = i/N, j' = j/i)$ which is not valid for $N = 0$ and $i = 0$. Hence point $(0, 0)$ is excluded from D_x : $D_x^* = D_x - \{(0, 0)\} = [1, N] \times [0, i]$.

On D_x^* , the polynomial $p(i, j)$ is first transformed into $(N^2 i'^2 + Ni')/2 + ij'$. Variable i' is then propagated and $\tilde{p}(i', j') = (N^2 i'^2 + Ni')/2 + Ni'j', (i', j') \in [0, 1] \times [0, 1]$.

□

We use this symbolic approach to get sufficient conditions for any considered parameterized multivariate polynomial to be strictly positive or negative. We show with some examples in next section how an appropriate instrumentation of this model allows automatic and inexpensive resolutions of complex program analysis issues.

6 Applications

A typical application of our mathematical framework is dependence testing between references with linearized subscripts. Linearized subscripts can result from optimizations like spatial data locality improvement [7]. Let us first consider an example given by Maslov and Pugh in [15] in order to show how our method can handle linearized subscripts. Although this example is also handled by Maslov and Pugh, we show in the following some cases that could not, because of the limitations of their method explained in section 7.

Example 3. The first example shown on figure 6 is a loop nest from the oil simulation program BOAST in the RiCEPS benchmark suite. Anyway this kind of loop nest is quite typical and met quite often in programs.

To be able to parallelize or to transform any of the i , j and k loops, it must be checked whether the flow dependence from the first statement to the second statement is loop-independent. Compilers fail to prove that this dependence is loop-independent.

We instrument our model in the following way: consider both iterations (i_1, j_1, k_1) and $(i_1 + \alpha, j_2, k_2)$. If we prove that:

$$MNi_1 + Nj_1 + k_1 < MN(i_1 + \alpha) + Nj_2 + k_2 \text{ for any } \alpha \geq 1$$

```

do i=p,p+L-1      do i=2,N          do i=1,N
  do j=q,q+M-1    do j=1,i-1      do j=i+1,(N-i)/2
    do k=r,r+N-1   if 4*i*i+j>=3*i*j+5*i+1  A(i,j)=A(j-1,N+i-j)
      A(M*N*i+N*j+k)=...  ...
      ...=A(M*N*i+N*j+k)  else
      enddo           ...
      enddo           endif
  enddo             enddo
enddo             enddo
enddo             enddo

```

(a) First typical example with a complex conditional statement loop nest for which acarized references.

(b) Second example with a complex conditional statement loop nest for which which can be eliminated.

(c) Third example of loop nest for which accessed array A has to be compacted.

Fig. 1. Examples.

then we prove that $MNi_1 + Nj_1 + k_1$ can never be equal to $MN(i_1 + \alpha) + Nj_2 + k_2$ and so that the dependence is independent of loop i . We use our method to get sufficient conditions for:

$$\begin{aligned} p(\bar{x}, \bar{u}) &= MNi_1 + Nj_1 + k_1 - (MN(i_1 + \alpha) + Nj_2 + k_2) \\ &= Nj_1 + k_1 - Nj_2 - k_2 - MN\alpha < 0 \end{aligned}$$

with $\bar{x} = (j_1, k_1, j_2, k_2)$ and $\bar{u} = (\alpha, N, M, p, L, q, r)$. Variables are considered on the box $D_x = [q, q + M - 1] \times [r, r + N - 1] \times [q, q + M - 1] \times [r, r + N - 1]$.

Transformation ϕ to map D_x on the unit box is defined by:

$$\bar{y} = \left(\frac{j_1 - q}{M - 1}, \frac{k_1 - r}{N - 1}, \frac{j_2 - q}{M - 1}, \frac{k_2 - r}{N - 1} \right)$$

which is valid for any variables values and for any parameters values strictly greater than one. It transforms polynomial $p(\bar{x}, \bar{u})$ into $\tilde{p}(\bar{y}, \bar{u})$:

$$\begin{aligned} \tilde{p}(\bar{y}, \bar{u}) &= N((M - 1)j'_1 + q) + (N - 1)k'_1 + r - N((M - 1)j'_2 + q) \\ &\quad - ((N - 1)k'_2 + r) - MN\alpha \\ &= (NM - N)j'_1 + (N - 1)k'_1 - (NM - N)j'_2 - (N - 1)k'_2 - MN\alpha \end{aligned}$$

with $\bar{y} = (j'_1, k'_1, j'_2, k'_2) \in [0, 1]^4$, $M > 1$ and $N > 1$. We use our software to compute the Bernstein coefficients of $\tilde{p}(\bar{y}, \bar{u})$ and get the following answer:

$$p(\bar{x}, \bar{u}) < 0 \text{ if } \begin{cases} NM\alpha > 0 \\ NM\alpha + N - 1 > 0 \\ NM\alpha + NM - N > 0 \\ NM\alpha + NM - 1 > 0 \\ NM\alpha - N + 1 > 0 \\ NM\alpha + NM - 2N + 1 > 0 \\ NM\alpha - NM + N > 0 \\ NM\alpha - NM + 2N - 1 > 0 \\ NM\alpha - NM + 1 > 0 \end{cases}$$

It can be easily checked that all these conditions are equivalent to $\alpha \geq 1$, since we are only concerned with integer values. Hence we have proved that the dependence is independent of loop i which can be parallelized. In the perspective of a fully automated framework, Bernstein expansion could be used again to check the nonlinear conditions.

□

We now consider an example showing our framework usability for dead code elimination of conditional statements.

Example 4. Consider the loop nest shown on figure 6. It contains a complex conditional statement that can be the result of induction variable recognition [16]. We use our tool to check whether the condition is either always true or always false in order to eventually eliminate some dead code.

Checking if $4i^2 + j \geq 3ij + 5i + 1$ is equivalent to check if $p(\bar{x}, \bar{u}) = 4i^2 - 3ij - 5i + j - 1 \geq 0$. Here we have $\bar{x} = (i, j)$, $\bar{u} = N$ and $D_x = [2, N] \times [1, i-1]$. Hence ϕ is defined by $\bar{y} = (\frac{i-2}{N-2}, \frac{j-1}{i-2})$ which is valid only if N is strictly greater than two and for all points excepting point $(2, 1)$: $D_x^* = [3, N] \times [1, i-1]$. It comes that N has to be strictly greater than three since ϕ is redefined as $\bar{y} = (\frac{i-3}{N-3}, \frac{j-1}{i-2})$. We compute $\tilde{p}(\bar{y}, \bar{u})$:

$$\begin{aligned}\tilde{p}(\bar{y}, \bar{u}) &= 4((N-3)i' + 3)^2 - 3((N-3)i' + 3)((i-2)j' + 1) - 5((N-3)i' + 3) \\ &\quad + ((i-2)j' + 1) - 1 \\ &= 4((N-3)i' + 3)^2 - 3((N-3)i' + 3)((((N-3)i' + 3) - 2)j' + 1) \\ &\quad - 5((N-3)i' + 3) + (((N-3)i' + 3) - 2)j' \\ &= (-24N + 36 + 4N^2)i'^2 + (-27 - 3N^2 + 18N)i'^2j' + (-48 + 16N)i' \\ &\quad + (-11N + 33)i'j' - 8j' + 12\end{aligned}$$

with $\bar{y} = (i', j') \in [0, 1]^2$ and $N > 3$. We compute the Bernstein coefficients of $\tilde{p}(\bar{y}, \bar{u})$ and get the following answer:

$$p(\bar{x}, \bar{u}) > 0 \text{ if } \{ 12 > 0, 4 > 0, 8N - 12 > 0, \frac{5N-7}{2} > 0, \\ 4N^2 - 8N > 0, N^2 - N - 2 > 0 \}$$

It can easily be checked that all these conditions always hold when $N > 2$. For point $(2, 1)$ which was excluded, we observe that $p(\bar{x}, \bar{u}) = 0$. Hence $p(\bar{x}, \bar{u})$ is always positive on D_x , the condition and the “else” parts of the code can be eliminated.

□

Example 5. This example aims to show that our polynomial evaluation method is also useful to achieve complex analysis and optimizations. Array contraction is an optimization that transforms an array into a smaller array within a loop in order to save memory and to increase locality. It is particularly useful for embedded systems.

Consider the loop nest shown on figure 6. We want to achieve contraction of array \mathbf{A} assuming that \mathbf{A} is a temporary array not further accessed in the rest of

the program. An array element is alive when it has been accessed through read or write at least once and will be accessed again in the future. The minimum amount of memory needed is given by the maximum number of simultaneously alive array elements. Contraction of array A to this amount can be achieved by defining new access functions and data layout.

Determining the maximum number of simultaneously alive array elements generally translates in maximizing a parameterized multi-variate polynomial. It is shown that such objective can be reached by our method.

We first compute the live range of any array element $A(x, y)$, i.e., the number of iterations occurring between the first and the last references to $A(x, y)$. The maximum live range will also be the maximum number of simultaneously alive array elements, and also the minimum amount of memory needed, since at each iteration an array element is accessed for the first time and another one is accessed for the last time. Moreover, each array element is referenced at most once by each reference.

The live range of $A(x, y)$ is computed in the following way: for each reference $A(i, j)$, $A(j-1, N+i-j)$ and $A(i-1, j-1)$, we compute the number of iterations occurring before the one where the considered reference resolves in $A(x, y)$. For example, considering reference $A(i, j)$, $A(x, y)$ is referenced when $i = x$ and $j = y$. Hence we compute the number of iterations occurring before iteration (x, y) included. When all the references have been considered, the live range of $A(x, y)$ is given by the largest difference between these iteration counts.

These iteration counts are given by the Ehrhart polynomials of some appropriately defined polytopes [7]. We get the following results:

Reference	Last Iteration	Iteration count
$A(i, j)$	(x, y)	$ic_1 = (N - \frac{1}{2})x - \frac{1}{2}x^2 + y - N$
$A(j-1, N+i-j)$	$(x + y - N + 1, x + 1)$	$ic_2 = -\frac{1}{2}x^2 - xy + (2N - \frac{1}{2})x - \frac{1}{2}y^2 + (2N - \frac{3}{2})y - \frac{3}{2}N^2 + \frac{3}{2}N$
$A(i-1, j-1)$	$(x + 1, y + x + 1)$	$ic_3 = xN - \frac{1}{2}x^2 - \frac{1}{2}x + y$

Since $A(j-1, N+i-j)$ and $A(i-1, j-1)$ do not reference the same array elements, since reference to $A(x, y)$ through $A(j-1, N+i-j)$ occurs before reference to $A(x, y)$ through $A(i, j)$, and since reference to $A(x, y)$ through $A(j-1, N+i-j)$ occurs after reference to $A(x, y)$ through $A(i, j)$, we compute the differences $ic_1 - ic_2 + 1$ and $ic_3 - ic_1 + 1$:

$$\begin{aligned} ic_1 - ic_2 + 1 &= xy - Nx + \frac{1}{2}y^2 + (\frac{5}{2} - 2N)y - \frac{5}{2}N + \frac{3}{2}N^2 + 1 \\ ic_3 - ic_1 + 1 &= N + 1 \end{aligned}$$

The maximum number of simultaneously active array elements that are referenced through $A(i-1, j-1)$ is $ic_3 - ic_1 + 1 = N + 1$. But in order to determine the maximum number of simultaneously active array elements that are referenced through $A(j-1, N+i-j)$, we need to determine the maximum value of $ic_1 - ic_2 + 1$ for any value of x and y . It is given by the largest Bernstein coefficient of $p(\bar{x}, \bar{u}) = ic_1 - ic_2 + 1$, where $\bar{x} = (x, y)$ and $\bar{u} = N$.

Since $1 \leq i \leq N$ and $i + 1 \leq j \leq (N - i)/2$, and since $x = j - 1$ and $y = N + i - j$, it comes that $1 \leq x \leq (N - 3)/2$ and $(N + 1)/2 \leq y \leq N - 1$.

Hence we have $\bar{x} = (x, y)$, $\bar{u} = N$ and $D_x = [1..(N-3)/2] \times [(N+1)/2..N-1]$. Transformation ϕ to map D_x on the unit box is defined by $\bar{y} = (\frac{2x-2}{N-5}, \frac{2y-N-1}{N-3})$ which is not valid for $N = 5$ and $N = 3$. Let us consider $N > 5$. Mapping ϕ transforms polynomial $p(\bar{x}, \bar{u})$ into $\tilde{p}(\bar{y}, \bar{u})$:

$$\begin{aligned}\tilde{p}(\bar{y}, \bar{u}) = & (\frac{1}{4}N^2 - 2N + \frac{15}{4})x'y' + (-\frac{1}{4}N^2 + \frac{3}{2}N - \frac{5}{4})x' + (\frac{1}{8}N^2 - \frac{3}{4}N + \frac{9}{8})y'^2 \\ & + (\frac{17}{4}N - 6 - \frac{3}{4}N^2)y' - \frac{5}{2}N + \frac{15}{8} + \frac{5}{8}N^2 + 1\end{aligned}$$

with $\bar{y} = (x', y') \in [0, 1]^2$ and $N > 5$. We compute the Bernstein coefficients of $\tilde{p}(\bar{y}, \bar{u})$ which are: $\frac{5}{8}N^2 - \frac{5}{2}N + \frac{23}{8}$, $\frac{1}{4}N^2 - \frac{3}{8}N - \frac{1}{8}$, $N - 2$, $\frac{3}{8}N^2 - N + \frac{13}{8}$, $\frac{1}{8}N^2 + \frac{1}{8}N + \frac{1}{2}$, $\frac{1}{2}N + \frac{1}{2}$.

The largest coefficient is obviously $\frac{5}{8}N^2 - \frac{5}{2}N + \frac{23}{8}$. Hence the maximum number of simultaneously active array elements that are referenced through $A(j-1, N+i-j)$ has been found, and the total number of simultaneously active array elements is $\frac{5}{8}N^2 - \frac{5}{2}N + \frac{23}{8} + N + 1 = \frac{5}{8}N^2 - \frac{3}{2}N + \frac{31}{8}$. Finally, array A can be contracted to $\frac{5}{8}N^2 - \frac{3}{2}N + \frac{31}{8}$ elements.

7 Limits, Further Developments, and Related Works

Two main problems may arise when applying our symbolic Bernstein method:

1. The program output can be a complex system of multivariate polynomial inequalities from which it is impossible to conclude immediately whether the initial polynomial inequation always holds or not, or what are the cases when it holds.
2. It can be impossible to conclude whether the initial polynomial is always positive or negative over the box D_x , since the output system of inequations has no solution.

In the first case, we propose to iterate our method on the most complex polynomials composing the output system, by setting a box for some parameters becoming the variables of a new problem. Solutions found by this way can then be propagated into the other polynomials forming the system. After several such iterations, solutions of the whole system should be found. We are currently investigating this approach.

In the second case, the initial box D_x needs to be cut in order to find solutions on some sub-boxes. Garloff presents in [12] a procedure to cut boxes in an opportune way. But since we consider parameterized polynomials, this procedure can not apply in our case. We will first try the approach of bisecting boxes into sub-boxes of equal volumes, and then think about a more elaborated strategy.

Another direction consists in elevating the degree of the polynomial $\tilde{p}(\bar{y}, \bar{u})$ over the unit box to make the Bernstein bounds tighter and get a system of inequations having some solutions.

Maslov and Pugh present in [15] a technique to simplify polynomial constraints. It is based on a decomposition of any polynomial constraint into a conjunction of affine constraints and 2-variable hyperbolic and elliptical inequalities and equalities that can later be linearized. Hence their approach is not

general and can only handle those polynomials that can be decomposed in this way.

In [6], Blume and Eigenmann propose to test dependences between iterations of a loop, in the presence of nonlinear expressions, by verifying whether the ranges of such an expression does not overlap between successive iterations. However their technique is only valid if the expressions are monotonically increasing or decreasing with the loop index, and if symbolic lower and upper bounds can be easily determined. We are not concerned with such limitation.

Same authors in [5] present a expression-comparison algorithm consisting in replacing each variable in expressions with their range. Their technique needs the use of rewrite rules for simplifying expressions containing ranges. Two variable replacing methods are described. A first one that can generate overly conservative lower and upper bounds, and a second one, giving more accurate bounds, but that only applies on expressions that are monotonic over the considered range. Moreover, monotony has first to be proved by use of the first method. Since overly conservative bounds can be generated, some monotonic expressions can improperly be evicted.

Van Engelen *et al.* in [9] propose to obtain accurate bounds of a polynomial over the parameterized box $[0, n - 1]$ by computing its Newton series. However their method is limited to univariate and monotonically increasing or decreasing polynomials.

8 Conclusion

It has been shown that the symbolic use of Bernstein expansion for the analysis of multivariate polynomials has several advantages:

- it is generally more accurate than classic interval methods, and even exact in many cases ;
- it handles any multivariate polynomial without restriction ;
- complexity of the output system of inequations directly depends on the complexity of the parameterized coefficients. If these coefficients are linear combination of parameters, the output is a system of linear equations that can be solved easily.

The main drawback is its need of computing time and memory which grows exponentially with the number of variables. However, most problems arising in static analysis involves only a few variables. Moreover, some very recent developments have shown the opportunity of a linear complexity algorithm [8].

Our symbolic approach has been implemented for the field of coefficients in \mathbb{Q} . The software uses the GNU-MP library for arbitrary precision arithmetic. For the future we plan to integrate our tool with the polyhedral library *PolyLib* in order to be able to directly solve the involved linear equation systems and to extend *Polylib* to handle multi-variate polynomials.

When considering nonlinear expressions in program analysis, propositions of non-exhaustive techniques become more and more current: program tracing,

heuristics, rough approximations, dynamic or run-time analysis, genetic algorithms, ... We argue that results obtained by these ways can never be as accurate as the ones obtained by static analysis using general mathematical frameworks. Moreover, correctness is becoming of paramount concern in more and more areas such as safety-critical systems or compilation into silicon.

References

1. The polyhedral library *polylib*. <http://icps.u-strasbg.fr/PolyLib>.
2. Jakob Berchtold and Adrian Bowyer. Robust arithmetic for multivariate bernstein-form polynomials. *Computer-aided Design*, 32:681–689, 2000.
3. S. Bernstein. *Collected Works*, volume 1. USSR Academy of Sciences, 1952.
4. S. Bernstein. *Collected Works*, volume 2. USSR Academy of Sciences, 1954.
5. W. Blume and R. Eigenmann. Symbolic range propagation. In *9th Int. Parallel Processing Symposium*, April 1995.
6. W. Blume and R. Eigenmann. Non-linear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1180–1194, December 1998.
7. Ph. Clauss and B. Meister. Automatic memory layout transformation to optimize spatial locality in parameterized loop nests. *ACM SIGARCH Computer Architecture News*, 28(1), March 2000.
8. J. Delgado and J.M. Peña. A linear complexity algorithm for the bernstein basis. In *IEEE Int. Conf. on Geometric Modeling and Graphics (GMAG'03)*, pages 162–167, July 2003.
9. R. Van Engelen, K. Gallivan, and B. Walsh. Tight timing estimation with the newton-gregory formulae. In *10th Workshop on Compilers for Parallel Computers, CPC 2003*, Jan. 2003.
10. G. Farin. *Curves and Surfaces in Computer Aided Geometric Design*. Academic Press, San Diego, 1993.
11. R.T. Farouki and V.T. Rajan. On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design*, 4:191–216, 1987.
12. J. Garloff. Application of bernstein expansion to the solution of control problems. In J. Vehi and M. A. Sainz, editors, *Proceedings of MISC'99 - Workshop on Applications of Interval Analysis to Systems and Control*, pages 421–430. University of Girona, Girona (Spain), 1999.
13. J. Garloff and B. Graf. *The Use of Symbolic Methods in Control System Analysis and Design*, chapter Solving Strict Polynomial Inequalities by Bernstein Expansion, pages 339–352. Institution of Electrical Engineers (IEE), London, 1999.
14. R. Martin, H. Shou, I. Voiculescu, A. Bowyer, and G. Wang. Comparison of interval methods for plotting algebraic curves. *Computer Aided Geometric Design*, 19:553–587, 2002.
15. Vadim Maslov and William Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. In *CONPAR 94 - VAPP VI, Int. Conf. on Parallel and Vector Processing*, Sept. 1994.
16. Sébastien Pop. Analysis of induction variables using chains of recurrences: extensions. Master's thesis, Université Louis Pasteur, Strasbourg, July 2003.
17. V. Stahl. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations*. PhD thesis, Johannes Kepler University Linz, Austria, 1995.

Periodic Polyhedra

Benoît Meister

LSIIT - ICPS
Pôle API
Bd Sebastian Brant
F-67400 Illkirch
`meister@icps.u-strasbg.fr`
`icps.u-strasbg.fr/~meister`

Abstract. This paper presents a new method for computing the integer hull of a parameterized rational polyhedron by introducing the concept of periodic polyhedron. Besides concerning generally parametric combinatorial optimization, the method has many applications for the analysis, optimization and parallelization of loop nests, especially in compilers.

1 Motivation

For many years, compiler writers have focused on parameterized loop nests, mainly because of their importance in scientific and multimedia programs. The *polytope model* [2] allows to manipulate loop nests whose bounds are affine functions with integer-valued parameters in the constant part by modeling them as parameterized rational polytopes P . As the considered loop indices are incremented by a constant integer value, the values taken by the n -vector of indices belong to a subset of \mathbb{Z}^n : an integer lattice L . So the values taken by the index vector $I \in \mathbb{Z}^n$, where n defines the number of nested loops, are given by the so-called \mathbb{Z} -polytope $P \cap L$.

Example 1. The iterations of the following Gaussian elimination code:

```
for(i=1;i<=n; i++)
    for(j=i+1;j<=n; j++)
        for(k=i+1; k<=n; k++)
            a[j][k]=a[j][k]-a[j][i]*a[i][k]/a[i][i];
```

are modeled by the parameterized \mathbb{Z} -polytope $P \cap \mathbb{Z}^3$, where

$$P = \{1 \leq i \leq n; i + 1 \leq j \leq n; i + 1 \leq k \leq n\}$$

and n is an integer parameter. □

As in the general problem of integer linear programming, we are interested in *integer points* (i.e., points with integer coordinates) in polytopes whose vertices may be non-integer.

In a class of code optimization and parallelization methods (e.g. in [20,12,21, 8,22]), loop nests are usually transformed by applying an affine integer transformation to the \mathbb{Z} -polytope representing the loop nest [17]. They may also be split

into sub-polyhedra as for example in [11]. The result can then be transformed back into a loop nest by source code generation.

Several loop optimization and parallelization techniques also need to compute the indices of the first and last iterations to be executed in a loop nest (i.e. the lexicographic extrema of the I values). For instance, a precise dependence analysis may consist in computing the first executed iteration accessing a variable, among the set of iterations following a given access to this variable. Another issue is precise liveness analysis of data accessed in a loop nest, where the first and last iterations accessing the considered data have to be computed. Liveness analysis can be used to reduce the maximum amount of memory used by a program, and to reduce communications while parallelizing loops.

These techniques, as well as integer programming techniques in general, look for vertices of the *integer hull* P' of a parameterized \mathbb{Z} -polytope, as it is known that the solution is one of these vertices. The integer hull is the convex hull of the integer points in $P \cap L$.

Many techniques exist in the literature for finding integer vertices. Following Nemhauser and Wolsey ([9] chap. II) and Aardal *et al* [1], they can be classified as follows:

- *cutting plane* algorithms compute the integer hull of a rational polyhedron by adding constraints that cut the non-integer vertices out of it, so that all its integral points still respect the added constraint. The performance of this method is strongly related to the choice of a new cutting plane. The original idea has been proposed by Gomory in [10]. Seminal algorithms and complexity studies are due to Chvatal [3] and Schrijver [18].
- *branch and bound* algorithms recursively divide P into sub-polyhedra of which integer vertices are computed. The most commonly used technique partitions P into two polyhedra by a hyperplane including the optimal rational vertex of P .
- algorithms based on Lagrangean relaxation give an approximate solution by integrating constraints into the objective function.
- other algorithms that iteratively find vertices of the integer hull (for example: column generation).

Note that these methods usually do not compute the integer hull of P but only some of its vertices. Moreover, since we are interested in exact solutions, the Lagrangean relaxation is not suitable.

In this paper, we present a new way to compute the integer hull P' of a rational parameterized polytope P , defined by a set of rational affine equalities and inequalities on a set of variables $I \in \mathbb{Z}^n$ and integer-valued parameters $N \in \mathbb{Z}^p$. Integer vertices are computed by recursively extracting, from the facets of P , sets of points having one more integer coordinate at each step.

Section 2 gives an intuitive explanation of the method. Section 3 then presents the class of mathematical objects to be used in this paper. The problem is then turned into a geometric form in section 4, which is used to devise an algorithm with some restrictions in section 5. These restrictions are eliminated in section 6. Finally, application of our method to find the integer lexicographic maximum

is presented in section 7, while section 8 gives concluding remarks and future works.

2 Background and Principle

We assume that the considered polytope P is defined by a set of non-redundant constraints. This is not a strong assumption, since any polyhedron can be defined by such constraints (see [18] chap. 8), which can eventually be computed [19]. Without loss of generality, we consider \mathbb{Z} -polyhedra whose supporting lattice is \mathbb{Z}^n .

Since P is parameterized, the shape of P depends on the value of the parameters. Wilde and Loehner have shown in [14,13] that the parameters space is partitioned into adjacent rational polyhedra, called *validity domains*, in which P has a given shape. Therefore, in a given validity domain, the coordinates of the vertices of P have a given affine expression. For simplicity, we first consider a polytope P whose parameters values belong to a unique validity domain. We show how to deal with the general case in section 6. Recall that we are concerned with integer-valued parameters. To have a better insight of the problem, let us consider the polytope presented in example 2.

Example 2. The following polytope is represented on figure 1.

$$P_2 = \{2i - 3j - 1 \geq 0; -i + 4j - 3 \geq 0; -2i + 25 \geq 0\}$$

□

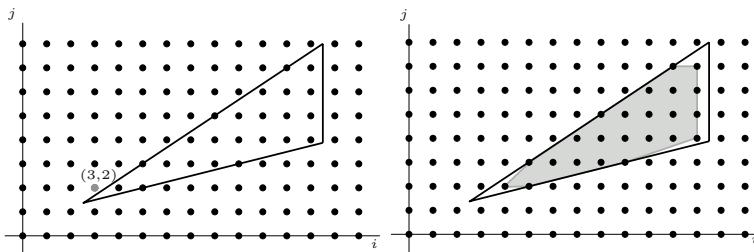


Fig. 1. P_2 , the false lexicographic minimum, and $\text{int}(P_2)$

Its corresponding loop nest in C, generated by using the Fourier-Moztskin algorithm, would be:

```
for (i=ceil(13/5); i<= floor(25/2); i++)
    for (j=ceil((i+3)/4); j<=floor((2i-1)/3); j++)
        /* statements */
```

According to the loop bounds, the minimal integer value for i is $\lceil 13/5 \rceil = 3$. The minimal integer value for j seems then to be $\lceil (i+3)/4 \rceil = \lceil 6/4 \rceil = 2$. But figure 1 shows that point $(3, 2)$ does not belong to P_2 : it does not correspond to a loop iteration. This example points out one of the problems one may encounter

when manipulating polyhedra with non-integer vertices while dealing with integer points. This problem vanishes if we manipulate the integer hull $\text{int}(P_2)$ of P_2 (in gray on figure 1), as its vertices are integer while it includes all the integer points (iterations) of P_2 .

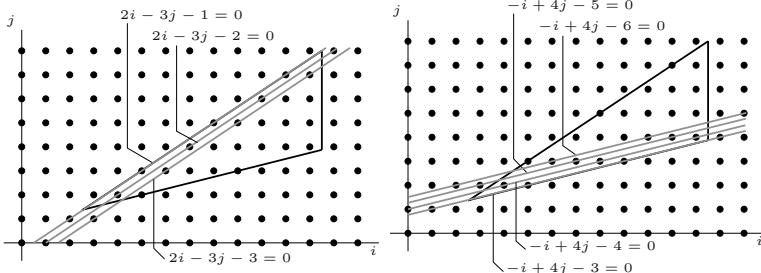


Fig. 2. Integer minima and maxima for j for a given value of i

Let us now consider the maximum integer value of j for a given i in P_2 . As one can see on figure 2, when $i \bmod 3 = 0$, the maximal integer value j_{max} is solution to the equation $2i - 3j - 3 = 0$. It is solution to $2i - 3j - 1 = 0$ when $i \bmod 3 = 2$, and solution to $2i - 3j - 2 = 0$ when $i \bmod 3 = 1$. Similarly, the minimal integer

value j_{min} is solution to $\begin{cases} -i + 4j - 4 = 0 & \text{if } i \bmod 4 = 0 \\ -i + 4j - 3 = 0 & \text{if } i \bmod 4 = 1 \\ -i + 4j - 6 = 0 & \text{if } i \bmod 4 = 2 \\ -i + 4j - 5 = 0 & \text{if } i \bmod 4 = 3 \end{cases}$. For a given value of

i , the coordinate in j of an integer point of P is bounded by values which are solutions to some equations. These equations depend periodically on i . So each integer point (i, j) of P_2 is in the convex hull of two integer *extremal* points: (i, j_{min}) and (i, j_{max}) . The convex hull of all these extremal points is then the integer hull of P_2 .

To generalize this idea, consider that any integer point

$$X = (x_1, x_2, \dots, x_n)$$

of P belongs to the set of points $d_k(X)$ of P defined by

$$d_k(X) = \{I = (x_1, \dots, x_{k-1}, i_k, x_{k+1}, \dots, x_n), i_k \in \mathbb{Q}, I \in P\}, k \in [1..n]$$

i_k has a minimal and maximal value in $d_k(X)$, given by two faces (or constraints) of P : $p_{min}(I, N) \geq 0$ and $p_{max}(I, N) \geq 0$. Coefficient a_k of i_k in $p_{min}(I, N)$ is positive and it is negative in $p_{max}(I, N)$, therefore we have: $i_k \geq p'_{min}(I, N)$ and $i_k \leq p'_{max}(I, N)$. These inequalities define the *rational* lower and upper bounds for i_k in $d_k(X)$. In other terms, we have $p'_{min}(I, N) \leq x_k \leq p'_{max}(I, N)$. Furthermore, as X_k is integer, we have $\lceil p'_{min}(X, N) \rceil \leq x_k \leq \lfloor p'_{max}(X, N) \rfloor$. So X belongs to the convex hull of the integer points $x_{min}(X, N)$ and $x_{max}(X, N)$, where

$$x_{min}(X, N) = (x_1, \dots, x_{k-1}, \lceil p'_{min}(X, N) \rceil, x_{k+1}, \dots, x_n)$$

and

$$x_{max}(X, N) = (x_1, \dots, x_{k-1}, \lfloor p'_{max}(X, N) \rfloor, x_{k+1}, \dots, x_n).$$

It follows that:

- the integer hull $int(P)$ of P is the convex hull of all the existing integer points $x_{min}(X, N)$ and $x_{max}(X, N)$,
- the vertices of $int(P)$ are some of these points.

Example 3. The existing points $x_{min}(i)$ and $x_{max}(i)$ are represented on figure 3, as well as the integer hull of P_2 and the different $d_2(i, j)$ (with $i = i_1$ and $j = i_2$). One can see that the convex hull of all these points is the integer hull of P . \square

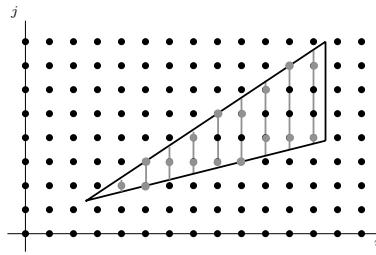


Fig. 3. The extremal integer points of P_2 and the different $d_2(i, j)$

We have seen that the extremal points are periodically solution to an equality. We explain this periodic character in next section.

3 Periodic Polyhedra

Consider the k^{th} coordinate of x_{max} , obtained from the inequality

$$p_{max}(I, N) = \sum_{t=1}^n a_t i_t + \sum_{t=1}^n b_t n_t + c \geq 0 : \\ x_{max,k} = \lfloor p'_{max}(X, N) \rfloor = \lfloor \sum_{t=1}^n a'_t x_t + \sum_{t=1}^p b'_t n_t + c' \rfloor$$

where $a'_k = 0$, $a'_t = a_t / (-a_k)$, $b'_t = b_t / (-a_k)$ and $c' = c / (-a_k)$ (a_k is negative). The common denominator of the rational coefficients of $p'_{max}(X, N)$ is $-a_k$. Let a be an integer and b a positive integer. We have $\lfloor \frac{a}{b} \rfloor = \frac{a}{b} - \frac{a \bmod b}{b}$. Let $p'_{max} = -a_k \cdot p'_{max}$. Then we can write: $x_{max,k} = \lfloor \frac{p'_{max}(X, N)}{-a_k} \rfloor$

$$x_{max,k} = \frac{p'_{max}(X, N)}{-a_k} - \frac{p'_{max}(X, N) \bmod (-a_k)}{-a_k}$$

$$-a_k x_{max,k} = p_{max}(X, N) - p_{max}(X, N) \bmod (-a_k)$$

As $p_{max}(X, N) + a_k x_k = p_{max}(X, N)$, this reduces to:

$$p_{max}(X, N) - p_{max}(X, N) \bmod (-a_k) = 0 \Leftrightarrow x_k = x_{max,k} \quad (1)$$

Similarly, we have:

$$p_{min}(X, N) - p_{min}(X, N) \bmod (a_k) = 0 \Leftrightarrow x_k = x_{min,k} \quad (2)$$

Equations (1) and (2) can be summed up into Theorem 1:

Theorem 1. Let $p(X, N) \geq 0$ be an inequality with integer coefficients. The value of x_k that is solution to

$$\mathcal{I}(p(X, N), i_k) = 0, \quad (3)$$

for integer values of $(x_1, x_2, \dots, x_{k-1}, x_{k+1}, \dots, x_n)$, is the extremal integer value for x_k given by $p(X, N) \geq 0$, where $\mathcal{I}(p(X, N), i_k) = p(X, N) - p(X, N) \bmod (|a_k|)$.

So the set of extremal points $x_{min}(X, N)$ and $x_{max}(X, N)$ are integer solutions to (3) for some constraint $p(X, N) \geq 0$ of P . In [16], we have shown that if $f(I)$ is an affine function with integer coefficients and m is integer, $f(I) \bmod m$ is an integer *periodic number*, with $I \in \mathbb{Z}^n$. A periodic number is a rational-valued periodic function of I (see [6,4]). It can be represented by a n -dimensional array whose number of elements in the k^{th} dimension is the corresponding period s_k .

Example 4. A 2-dimensional periodic number of period $S = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$ depending on $(N, M) \in \mathbb{Z}^2$, whose value is :

- 1 for $N \bmod 2 = 0, M \bmod 3 = 0$
- 2 for $N \bmod 2 = 1, M \bmod 3 = 0$
- 3 for $N \bmod 2 = 0, M \bmod 3 = 1$
- 4 for $N \bmod 2 = 1, M \bmod 3 = 1$
- 5 for $N \bmod 2 = 0, M \bmod 3 = 2$
- 0 for $N \bmod 2 = 1, M \bmod 3 = 2$

can be represented as: $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 0 \end{bmatrix}_{N,M}$ □

Consider $A(i) = i \bmod 3, i \in \mathbb{Z}$. Its value is periodically 0, 1, and 2, depending on i . Hence it is a periodic number of period 3. In a more general way, if $f(I) = \sum_{k=1}^n a_k i_k + c$ is an integer affine function of $I \in \mathbb{Z}^n$ with $a_k, c \in \mathbb{Z}$, and $m \in \mathbb{N}$, $f(I) \bmod m$ is a periodic integer number of period $S = (s_k)$ with $s_k = \frac{m}{\gcd(m, |a_k|)}$.

Example 5. With $f(I) = 3i + 4j$, and $m = 6$, it gives:

$$(3i + 4j) \bmod 6 = [0 \ 1 \ 2 \ 3 \ 4 \ 5]_{(3i+4j)} = \begin{bmatrix} 0 & 4 & 2 \\ 3 & 1 & 5 \end{bmatrix}_{i,j} \quad \square$$

We can then give an explicit form of the equalities yielding the extremal points by using periodic numbers.

Example 6. According to theorem 1, the j coordinate of the extremal points of P_2 is given by one of the two equations:

$$-i + 4j - 4 - (-i + 4j - 3) \bmod 4 = -i + 4j - 3 - [1 \ 0 \ 3 \ 2]_i = 0$$

$$2i - 3j - 1 - (2i - 3j - 1) \bmod 2 = 2i - 3j - 1 - [1 \ 0]_i = 0,$$

which is concordant with our preliminary observations. \square

These equalities depend on a periodic number: their *definition* is periodic. As a classical equality defines a hyperplane, i.e. a polyhedron, an equality whose definition is periodic defines then a periodic polyhedron. In [16], we present a general class of mathematical objects depending periodically on multiple parameters, which are called *periodics*.

Definition 1. A n -dimensional periodic polyhedron M of period $S = (s_1 \ s_2 \ \dots \ s_n)^T \in \mathbb{N}^n$ is given by:

- $q = s_1 \times s_2 \times \dots \times s_n$ polyhedra M_I , indexed by:

$$I = (i_1 \ i_2 \ \dots \ i_n)^T \in \mathbb{Z}^n \text{ with } 0 \leq i_k < s_k, k \in [1..n]$$

- their respective definition domain \mathcal{D}_I : the integer lattice defined by:

$$\begin{pmatrix} s_1 & 0 & \cdots & 0 \\ 0 & s_2 & 0 & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & s_m \end{pmatrix} J + I, J \in \mathbb{Z}^n$$

$\bigcup_I \mathcal{D}_I = \mathbb{Z}^n$: any element $X \in \mathbb{Z}^n$ is mapped to a unique M_I by the relation $I = X \bmod S$. \square

Notice that each M_I is then the intersection between an integer lattice and a polyhedron (i.e. a \mathbb{Z} -polyhedron).

Example 7. A 2-dimensional periodic over polyhedra (made of one inequality) of $(i, j, k) \in \mathbb{Z}^3$ depending on $(i, j) \in \mathbb{Z}^2$, whose value is :

- $i + 2j - 3k + 7 \geq 0$ for $i \bmod 3 = 0, j \bmod 2 = 0$
- $2i - j + k \geq 0$ for $i \bmod 3 = 1, j \bmod 2 = 0$
- $3j - 5 \geq 0$ for $i \bmod 3 = 2, j \bmod 2 = 0$
- $-i - 2j \geq 0 + 3k - 7$ for $i \bmod 3 = 0, j \bmod 2 = 1$
- $i - 2k + 1 \geq 0$ for $i \bmod 3 = 1, j \bmod 2 = 1$
- $1 \geq 0$ for $i \bmod 3 = 2, j \bmod 2 = 1$

can be represented as:

$$\begin{bmatrix} i + 2j - 3k + 7 \geq 0 & -i - 2j + 3k - 7 \geq 0 \\ 2i - j + k \geq 0 & i - 2k + 1 \geq 0 \\ 3j - 5 \geq 0 & 1 \geq 0 \end{bmatrix}_{i,j}, \text{ and can also be written:}$$

$$\begin{bmatrix} 1 & -1 \\ 2 & 1 \\ 0 & 0 \end{bmatrix}_{i,j} i + \begin{bmatrix} 2 & -2 \\ -1 & 0 \\ 3 & 0 \end{bmatrix}_{i,j} j + \begin{bmatrix} -3 & 3 \\ 1 & -2 \\ 0 & 0 \end{bmatrix}_{i,j} k + \begin{bmatrix} 7 & -7 \\ 0 & 1 \\ -5 & 1 \end{bmatrix} \geq 0$$

(we call this form *factorized form*, as the constant form of an affine function of (i, j, k) is extracted). \square

Using the knowledge of periodic polyhedra, we show in the next section new geometric tools that will allow to give an algorithm for computing the integer hull of a polyhedron.

4 Pseudo-Facets

We have seen that extremal points of P are integer solutions to an equality $\mathcal{I}(p(X, N), i_k) = 0$ where $p(X, N) \geq 0$ is one of the inequalities that define P . As well as the inequality $p(X, N) \geq 0$ defines a facet of P (with its adjacent faces), we say that the solutions to $\mathcal{I}(p(X, N), i_k) = 0$ belong to a *pseudo-facet* of P .

Definition 2. The q^{th} facet $f_q(P)$ of a polyhedron P can be defined by: $\{I \in P \mid p_q(I, N) = 0\}$. Similarly, the q^{th} pseudo-facet $f'_{q,k}(P)$ of P w.r.t. i_k is defined by: $\{I \in P \mid \mathcal{I}(p_q(I, N), i_k) = 0\}$.

A pseudo-facet $f_q(P)$ can be decomposed into:

- a (*supporting*) *pseudo-hyperplane*, defined by $\mathcal{I}(p_q(I, N), i_k) = 0$,
- a *projected pseudo-facet*, which is the projection of the pseudo-facet along i_k by using the equality $\mathcal{I}(p_q(I, N), i_k) = 0$.

The projected pseudo-facet defines the values of all the variables but i_k for which the solution of $\mathcal{I}(p_q(I, N), i_k) = 0$ belongs to the pseudo-facet $f'_{q,k}(P)$.

Example 8. Consider the parameterized polyhedron P_3 defined by:

$$P_3 = \{-2i_1 + 3i_2 - n \geq 0; -2i_2 + 21 \geq 0; 4i_1 + i_2 - 13 \geq 0\}$$

Figure 8 shows the facet of P defined by:

$$\begin{cases} -2i_1 + 3i_2 - n = 0 \\ -2i_2 + 21 \geq 0 \\ 4i_1 + i_2 - 13 \geq 0 \end{cases} = \begin{cases} -2i_1 + 3i_2 - n = 0 \\ \begin{cases} -2i_2 + 21 \geq 0 \\ 7i_2 - 2n - 13 \geq 0 \end{cases} \end{cases}$$

and the corresponding pseudo-facet w.r.t. i_1 :

$$\begin{cases} \mathcal{I}(-2i_1 + 3i_2 - n, i_1) = -2i_1 + 3i_2 - n - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_{i_2,n} = 0 \\ -2i_2 + 21 \geq 0 \\ 4i_1 + i_2 - 13 \geq 0 \end{cases}$$

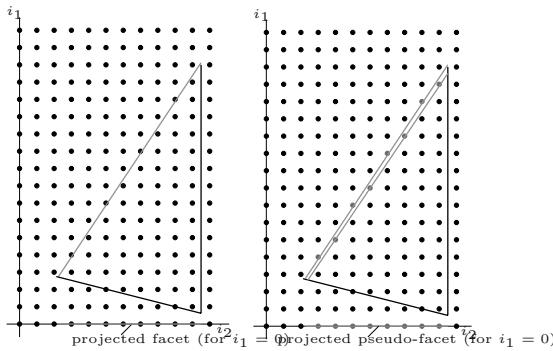


Fig. 4. A facet of P_3 and its corresponding pseudo-facet w.r.t. i_1 .

$$= \begin{cases} -2i_1 + 3i_2 - n - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_{i_2, n} = 0 \\ \{-2i_2 + 21 \geq 0; 7i_2 - 2n - \begin{bmatrix} 13 & 15 \\ 15 & 13 \end{bmatrix}_{i_2, n} \geq 0\} \end{cases}$$

□

This allows a more geometrical expression of the theorem 1, as the integer points whose convex hull is the integer hull of P belong to the pseudo-facets of P w.r.t. i_k .

Theorem 2. *The convex hull of the integer hulls of the pseudo-facets of P w.r.t. i_k is the integer hull of P .*

Proof. Any (integer) extremal point belongs to the integer hull of a pseudo-facet of P w.r.t. i_k . The convex hull of all the existing integer hulls is then the integer hull of the extremal points, that is to say the integer hull of P .

The determination of the integer hull of P therefore requires to compute the integer hull of each of the pseudo-facets of P w.r.t. i_k . But a pseudo-facet is a periodic polyhedron. We have shown in [15] that the pseudo-facet of a periodic polyhedron P' w.r.t. variable i_k is computed by taking into account the period of P along i_k . The supporting hyperplane is given by:

$$\mathcal{I}\left(\sum_l a_l i_l + c, i_k\right) = \sum_l a_l i_l + c - ((\sum_l a_l i_l + c) \bmod |a_k s_k|),$$

where s_k is the period of P' along i_k . Computing pseudo-facets for pseudo-polyhedron allows us to compute the integer hull of a pseudo-facet. This is used by the algorithm for computing the integer hull of a parameterized rational polyhedron, as presented in the next section.

5 Integer Hull

Theorem 2 states a relation between the integer hull $\text{int}(P)$ of polyhedron P and the integer hull of its pseudo-facets w.r.t. variable i_k :

$$\text{int}(P) = \text{conv}(\bigcup_q \text{int}(f'_{q,k}(P)))$$

The problem is now to compute $\text{int}((f'_{q,k}(P)))$. Similarly to the integer hull of P , the integer hull of $f'_{q,k}$ is the integer hull of its pseudo-facets w.r.t. another variable $i_{k'}, k' \in [1..n] \setminus k$ as the points of $f'_{q,k}$ correspond already to an integer bound of i_k . Thus we have:

$$\text{int}((f'_{q,k})) = \text{conv}(\bigcup_{q'} \text{int}(f''_{q',k'}(f'_{q,k}))) \quad (4)$$

This establishes a recurrence relation between the integer hull of a pseudo-facet and the integer hull of its own pseudo-facets. According to theorem 1, a property of a pseudo-facet w.r.t. i_k is that i_k is integer if the other variables and the parameters are integer. A pseudo-facet f'' w.r.t $i_{k'}$ of a pseudo-facet f' w.r.t. i_k belongs to f' : i_k is integer if the other variables and the parameters are integer. Moreover, it is a pseudo-facet w.r.t. $i_{k'}$, so $i_{k'}$ is integer if the other variables and the parameters are integer. Then i_k and $i_{k'}$ are integer if the other variables are integer. Recursively, taking n times a pseudo-facet of a pseudo-facet leads to a pseudo-facet f such that all the variables are integer when the parameters are integer. Since a pseudo-facet of a periodic polyhedron of dimension m is $(m-1)$ -dimensional, f is of dimension 0. The two latter sentences state that f is an integer point. The integer hull of P is then the convex hull of all the f 's. The relations among P , its pseudo-facets and recursively the pseudo-facets of the pseudo-facets are given by a tree, the *pseudo-facet tree* P . The vertices of $\text{int}(P)$ are obtained by scanning this tree from its root P to its 0 -dimensional pseudo-facets (the vertices), called the *pseudo-vertices*. We can devise an algorithm from the recurrence relation among P and its pseudo-facets to obtain the pseudo-vertices of P .

```
get_pseudo_vertices(periodic polyhedron P) {
    n = dimension(P)
    if n = 0 return P
    U = empty set of periodic polyhedra
    k = rank of the variable to be processed for n
    i = the kth variable of V
    for each inequality f(I) >= 0 of P
        with a nonzero coefficient for i do :
            compute P' by replacing f(I) >= 0 in P
            by  $f(I) - (f(I) \bmod |a|) = 0$ 
            ==>  $P' = \{P \mid f(I) - (f(I) \bmod |a|) = 0\}$ 
            P_proj= P' projected pseudo-facet of P,
            add get_pseudo_vertices(P_proj) to U
            for each element u of U
                add  $\{f(I) - (f(I) \bmod |a|) = 0\} \cap u$  to P'
            endfor
    endfor
    return P'
}
```

By opposition to cutting planes algorithms, our algorithm adds no constraint to the original problem, but constraints are replaced by periodic constraints. It adds neither variables nor parameters to the problem. Moreover, the pseudo-facet tree of P is n -deep, so a given pseudo-vertex is obtained in computing a pseudo-facet n times.

However, it handles periodic numbers, which can have large periods if the coefficients of the constraints of P are large. An upper bound of the number of elements of the computed periodics is M^{n+p} , where M is the maximal value of a coefficient that can be obtained by row elimination of the constraints of P , and n and p are the number of variables and parameters. One can still consider the use of the symbolic expression of a periodic number, especially when M is large, to achieve the computation.

Computing the pseudo-vertices of a n -dimensional simplex requires $(n+1)!$ computations of a pseudo-facet, which gives a lower complexity bound. Notice that the algorithm is highly parallel: the processing of subtrees can be distributed, and the only data to be communicated is the pseudo-facet corresponding to the subtree root.

6 Generalization

6.1 Unbounded Polyhedra

Unbounded polyhedra can be used to model nested *while* and *for* loops with the polytope model, as for instance in [5]. Nemhauser and Wolsey have proved in [9](part I, section 6) that the integer hull S of an unbounded polyhedron P defined by its Minkowsky representation:

$$P = \{x \in \mathbb{R}^n \mid x = \sum_k \lambda_k v^k + \sum_j \mu_j r^j, \sum_k \lambda_k = 1, \mu_j \geq 0 \forall j\}$$

where the $v^k \in \mathbb{R}^n$ are the extreme vertices of P and r^j its extreme rays, is given by:

$$S = \{x \in \mathbb{Z}^n \mid x = \sum_k \lambda_k q^k + \sum_j \mu_j r^j, \sum_k \lambda_k = 1, \mu_j \geq 0 \forall j\}$$

where the q^k are the extremal vertices of the integer hull of $\text{conv}(v^k)$. Thus, computing the integer hull of an unbounded polyhedron P amounts to compute the integer hull of its extreme vertices.

6.2 Polyhedron with Multiple Validity Domains

In the general case of a parameterized polyhedron, the computed pseudo-vertices will actually belong to the polyhedron for only some values of the parameters. Loechner has given in [13] an algorithm to compute the values of parameters for which a given rational vertex belongs to a rational polyhedron. The parameters space is then partitioned into polyhedral domains in which a given set of vertices

belong to P . This method can be adapted directly to compute pseudo-vertices: we can partition the parameter space into periodic polyhedra in which a given set of pseudo-vertices belong to P . This is translated into the following algorithm:

```
get_integer_hull(Polyhedron P) {
    IH = empty list of periodic polyhedra
    PX = empty list of periodic polyhedra
    PX = get_pseudo_vertices(P)
    (V, X) = lists of validity domains and
              the corresponding valid pseudo-vertices from PX
    for each (V_i, X_i) in (V, X)
        H_i = convex hull of the pseudo-vertices X_i
        add H_i to H
    endfor
    return (V, H)
}
```

6.3 Non-fully-dimensional Polyhedra

The geometric dimension of a polyhedron P defined in a n -dimensional variable space by a non-redundant system of e equalities and e' inequalities is given by $d(P) = n - e$. A polyhedron is fully-dimensional if $d(P) = n$, so if P has no equalities. Integer solutions to a system of e equalities belong to a $(n - e)$ -dimensional integer lattice, whose determinant may be greater than 1 and which depends on the parameters. Methods to deal with this case will be presented in a further work.

7 An Application: Computing the Integer Lexicographic Extremum of a Polytope

The lexicographic order is a total order on the elements of a n -dimensional space. It is noted \triangleleft and defined by:

$$I = (i_1 \ i_2 \ \dots \ i_n) \in \mathbb{Z}^n \triangleleft J = (j_1 \ j_2 \ \dots \ j_n) \in \mathbb{Z}^n$$

$$\Leftrightarrow \exists k \in [1..n] : i_1 = j_1, i_2 = j_2 \dots i_{k-1} = j_{k-1}, i_k < j_k$$

For loop nests in computer programs, the integer lexicographic minimum (resp. maximum) in $P \cap \mathbb{Z}^n$ corresponds to the first (resp. last) iteration to be executed. Feautrier presents in [7] a method to find the lexicographic extrema of a parameterized loop nest, based on parameterized versions of Gomory's fractional cutting planes approach and the lexicographic dual simplex algorithm. Both integer lexicographic maximum or minimum are obtained in a similar way, roughly by inverting comparison and extrema operators. So we focus on the computation of the integer lexicographic maximum. The integer lexicographic maximum of P belongs to the set of integer points X_1 of $\mathbb{Z}^n \cap P$ having the maximal existing integer value for i_1 . If such a set contains several points, the integer lexicographic maximum is in the subset of X_1 having the maximal integer value for i_2 , and so on. As for any solution to a linear integer programming problem, we know that the lexicographic maximum is a vertex of the integer hull of P .

A straightforward way to compute the integer lexicographic maximum of P would therefore be to compute its integer hull P' and then to find the lexicographic rational maximum in P' , for example by using the lexicographic dual simplex algorithm (see [7] or [9] chap. II section 4). But this approach yields the computation of every vertex, which is costly and in general useless as some vertices can never be the lexicographic maximum. So, as it is generally done in integer linear programming, we must compute only a subset of the vertices of P' , containing the integer lexicographic maximum of P . We can equivalently compute the set of vertices of P' that excludes a maximal number of vertices that cannot be the lexicographic maximum. The efficiency of the method depends on the selection of the pseudo-facets that will contain the candidate vertices.

The maximal *rational* values of i_1 are those of the faces of P which are upper bounds for i_1 : those whose coefficient on i_1 is negative. Candidate vertices therefore belong to the corresponding pseudo-facets f of P w.r.t. i_1 , which eliminates the vertices that are in the other pseudo-facets. From now on, we will call *min-pseudo-facet* (resp. *max-pseudo-facet*) of a periodic polyhedron P w.r.t. i_k a pseudo-facet corresponding to an integer lower (resp. upper) bound of i_k in P .

Example 9. The lexicographic maximum of P_3 belongs to its only max-pseudo-facet w.r.t. i_1 , defined by:

$$f(I, N) = \begin{cases} -2i_1 + 3i_2 - n - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_{i_2, n} = 0 \\ \{-2i_2 + 21 \geq 0; 7i_2 - 2n - \begin{bmatrix} 13 & 15 \\ 15 & 13 \end{bmatrix}_{i_2, n} \geq 0\} \end{cases}$$

□

Let $a_1 i_1 + a_2 i_2 + \dots + a_n i_n + a_0 = 0$ be the associated supporting pseudo-hyperplane. For a given value of the parameters, maximizing i_1 is equivalent to maximize $g(I) = a_2 i_2 + \dots + a_n i_n$, since $i_1 = -\frac{a_2 i_2 + \dots + a_n i_n}{a_1}$ and $a_1 < 0$. This is a second elimination criterion, which allows to keep the pseudo-facets of f that can contain the integer points with a maximal value for $g(I)$. Let us see two ways to select relevant pseudo-facets of f .

First, we can consider the variation of $g(I)$ w.r.t. the sign of a_k . $g(I)$ increases with variables i_{inc} with $a_k > 0$. Thus, the candidate vertices belong to pseudo-facets that maximize such i_{inc} . But $g(I)$ also decreases with variables i_{dec} with a negative coefficient and is constant for i_{cst} with a zero coefficient. In the same way, we will also consider pseudo-facets that minimize i_{dec} . Besides, we have to include max-pseudo-facets as well as min-pseudo-facets w.r.t. i_{cst} .

Example 10. In our example, i_2 increases in function of i_1 , so the lexicographic maximum belongs to the max-pseudo-facets of $f(I, N)$ w.r.t. i_2 : $\{-2i_1 + 3i_2 - n - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_{i_2, n} = 0; -2i_2 + [20 18]_{i_2} = 0\}$. The candidate pseudo-vertex is then:

$$(i_1, i_2) = ((-n + \begin{bmatrix} 30 & 29 \\ 26 & 27 \end{bmatrix}_{i_2, n})/2, [10 9]_{i_2}).$$

For each value of n , the maximal

value of i_1 is reached for $i_2 \bmod 2 = 0$. The integer lexicographic maximum is then: $(i_1, i_2) = (-n + [30 29]_n)/2, 10)$ \square

This pseudo-facet selection method can be written as an algorithm, detailed in [15].

In general, each pseudo-vertex depends periodically on the variables and the parameters. But there is always one optimal value of the variables for each value of the parameters, which must be selected.

Example 11. Let us make a sketch of the process to find the candidate vertices for a lexicographic maximum with the following example (also considered by Feautrier in [7]): $P = \{-i + m \geq 0; i \geq 0; j \geq 0; -j + n \geq 0; 2i + j - k = 0\}$ where k, n, m are the parameters. Since there is an equality, for a given (maximal) i, j is determined, so we can eliminate j . The coefficient in j is 1, so j is integer for any integer value of the variables and the parameters. Therefore, the problem presented in section 6.3 does not occur here.

$$P = \left\{ \begin{array}{l} 2i + j - k = 0 \\ \{-i + m \geq 0; i \geq 0; -2i + k \geq 0; 2i - k + n \geq 0\} \end{array} \right.$$

The candidate vertices belong to two max-pseudo-facets of P w.r.t. i :

$$A = \left\{ \begin{array}{l} 2i + j - k = 0 \\ -i + m = 0 \\ i \geq 0 \\ -2i + k \geq 0 \\ 2i - k + n \geq 0 \end{array} \right. = \left\{ \begin{array}{l} 2i + j - k = 0 \\ -i + m = 0 \\ m \geq 0 \\ k - 2m \geq 0 \\ -k + 2m + n \geq 0 \end{array} \right.$$

$$B = \left\{ \begin{array}{l} 2i + j - k = 0 \\ -i + m \geq 0 \\ i \geq 0 \\ -2i + k - (-2i + k) \bmod 2 = 0 \\ 2i - k + n \geq 0 \end{array} \right. = \left\{ \begin{array}{l} 2i + j - k = 0 \\ -2i + k - [0 1]_k = 0 \\ -k + 2m + [0 1]_k \geq 0 \\ k - [0 1]_k \geq 0 \\ n - [0 1]_k \geq 0 \end{array} \right.$$

Here, A and B are the pseudo-vertices we are looking for:

$$A : \binom{i}{j} = \binom{m}{k - 2m} \text{ for } \left\{ \begin{array}{l} m \geq 0 \\ 2m \leq k \leq 2m + n \end{array} \right.$$

$$B : \binom{i}{j} = \binom{k - [0 1]_k}{[0 1]_k} \text{ for } \left\{ \begin{array}{l} n \geq [0 1]_k \\ [0 1]_k \leq k \leq 2m + [0 1]_k \end{array} \right.$$

\square

In the general case, the domains of parameters are unions of polyhedra. As the algorithm computes extrema one variable at a time, the result can take the form of a quasi-affine syntax tree (as in Feautrier's PIP) or of a union of polyhedra.

Another pseudo-facet selection method would be to select the pseudo-facets corresponding to faces of equation $\alpha_1 i_1 + \alpha_2 i_2 + \dots + \alpha_n i_n + \alpha_0$ such that $(0 \ a_2 \ \dots \ a_n) \cdot (\alpha_1 \ \alpha_2 \ \dots \ \alpha_n)^T \leq 0$. Such faces are rational maximal bounds for $g(I)$, which is easily figured out by a rational linear transformation.

8 Conclusion

We have defined a new method for computing the integer hull of a rational parameterized polyhedron, which has many applications in loop nests analysis and optimization. This geometric method also yields new ways to solve parametric integer programming problems. In particular, it can help solving integer lexicographic extrema, which is an omnipresent problem in loop nest analysis, optimization and parallelization. One of our objectives was to provide a method based on the underlying concepts of the polyhedral library Polylib [19]. While considering this objective, we particularly generalized the concept of periodic numbers to periodic polyhedra. The presented algorithms are being implemented in the library. Their performance will be compared with the existing solvers. Other problems can also be tackled using our method: the domain of parameters for which an integer point exists in a polyhedron, the number of integer points in the projection of a \mathbb{Z} -polyhedron, and alternative ways to compute Ehrhart polynomials are currently worked on. We also plan to write a generic parametric integer solver, by finding better pseudo-facet elimination criteria, and to compare our methods *in silico* with the existing ones.

References

1. K. Aardal and S. van Hoesel. Polyhedral techniques in combinatorial optimization ii: Computations. *Statistica Neerlandica*, 50:3, 1996.
2. C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. ACM SIGPLAN '91*, pages 39–50, june 1991.
3. V. Chvatal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4:305–337, 1973.
4. Ph. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *10th ACM Int. Conf. on Supercomputing, Philadelphia*, 1996.
5. J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *proc. of the Scalable High-Performance Computing Conf.(SHPCC'94)*, pages 429–436. IEEE, May 1994.
6. E. Ehrhart. *Polynômes arithmétiques et Méthode des Polyèdres en Combinatoire*, volume 35 of *International Series of Numerical Mathematics*. Birkhäuser Verlag, Basel/Stuttgart, 1977.
7. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.
8. P. Feautrier. Lncs 1132. In *The Data Parallel Programming Model*, chapter Automatic parallelization in the Polytope Model, pages 79–100. 1996.
9. G.Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. W-I Series in Discrete Mathematics and Optimization. Wiley-Interscience, 1988. ISBN 0-471-82819-X.
10. R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the AMS*, 64:275–278, 1958.
11. Martin Griebl, Paul A. Feautrier, and Christian Lengauer. Index set splitting. *Int. J. Parallel Programming*, 28(6):607–631, 2000.
12. C. Lengauer. Loop parallelization in the polytope model. In *CONCUR 93*, 1993.

13. V. Loechner. *Contribution l' tude des poly dres param tr s et applications en parallelisation automatique*. PhD thesis, Universit Louis Pasteur, 1997. <http://icps.u-strasbg.fr/pub-97/>.
14. V. Loechner and D. K. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6):525–549, Dec. 1997.
15. B. Meister. Periodic polyhedra. research report, LSIIT - ICPS, October 2003.
16. B. Meister. Using periodics in integer polyhedral problems. research report, LSIIT - ICPS, 2003. <http://icps.u-strasbg.fr/~meister>.
17. P. Quinton, S. V. Rajopadhye, and T. Risset. On manipulating z-polyhedra using a canonical representation. *Parallel Processing Letter*, 7(2):181–194, 1997.
18. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1986.
19. D.K. Wilde. A library for doing polyhedral operations. Technical report 785, IRISA, Rennes, France, 1993.
20. M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
21. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.
22. J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, 1997.

Region-Based Partial Dead Code Elimination on Predicated Code

Qiong Cai¹, Lin Gao^{2†}, and Jingling Xue^{1‡}

¹ School of Computer Science and Engineering, University of New South Wales,
Sydney, NSW 2052, Australia

² Institute of Computing Technology, Chinese Academy of Sciences

Abstract. This paper presents the design, implementation and experimental evaluation of a practical region-based partial dead code elimination (PDE) algorithm on predicated code in an existing compiler framework. Our algorithm processes PDE candidates using a worklist and reasons about their partial deadness using predicate partition graphs. It operates uniformly on hyperblocks and regions comprising basic blocks and hyperblocks. The result of applying our algorithm to an SEME region is optimal: partially dead code cannot be removed without changing the branching structure of the program or potentially introducing new predicate defining instructions. We present statistic evidence about the PDE opportunities in the 12 SPECint2000 benchmarks. In addition to exhibit small compilation overheads, our algorithm achieves moderate performance improvements in 8 out of the 12 benchmarks on an Itanium machine. Our performance results and statistics show the usefulness of our algorithm as a pass applied before instruction scheduling.

1 Introduction

EPIC architectures such as the Intel IA-64 combine explicit instruction-level parallelism (ILP) with instruction predication. In order to realise their performance potential, the compiler must expose and express increasing amounts of ILP in application programs. Region-based compilation as proposed in [12] and implemented in the IMPACT [8] and ORC [19] compilers repartitions the program into regions to replace functions as the units of compilation. By exploiting the static/dynamic profile information, the compiler can successfully create regions that more accurately reflect the dynamic behavior of the program. In particular, by forming regions containing cycles across the function boundaries, the potential for the compiler to expose more ILP is increased [12,20]. By selecting the sizes and contents of regions appropriately, the compiler can also tradeoff compilation cost and the use of aggressive ILP techniques.

[†] Lin Gao is a Visiting Research Associate at the University of New South Wales since February 2003. She is supported by an ARC Grant DP0211793.

[‡] Jingling Xue gracefully acknowledges Intel's donation of Itanium computers.

Predicated execution, supported by EPIC architectures, allows parallel execution of instructions from multiple control paths and aids in efficient instruction scheduling. In this architectural model, an instruction may be guarded by a boolean predicate, which determines whether the instruction is executed or nullified. The values of predicates are manipulated by a set of predicate defining instructions. To explore predication, the compiler generally uses a technique called if-conversion [1], which eliminates branch instructions and replaces affected instructions with predicate defining instructions and predicated forms. This technique enlarges the scope of instruction scheduling and avoids branch misprediction penalties that may be caused by the eliminated branches.

This paper discusses the design, implementation and experimental evaluation of a new, practical region-based partial dead code elimination (PDE) algorithm on predicated code. An assignment is *partially dead* if there is a path along which the value computed by the assignment is never used, and is *fully dead* if it is partially dead along all such paths. Dead code appears frequently as a result of the optimisations applied earlier, including partial redundancy elimination (PRE), strength reduction and global copy propagation. PDE, which subsumes the dead code elimination (DCE), is an aggressive global optimisation. It is harder than PRE due to the second-order effects clearly exemplified in [18].

Existing PDE algorithms [5,11,18] are developed for non-predicated code assuming functions as compilation units. These algorithms are inadequate when instructions are predicated. Consider the following code snippet:

```
x = a + b (p1)
y = x + 1 (p2)
x = c - d (p3)
```

Being insensitive to the three predicates, classic PDE algorithms cannot determine if the first assignment is partially or fully dead or even dead at all.

The contributions of this work are summarised as follows:

- We introduce a practical region-based PDE algorithm on predicated code. By using the notion of Predicate Partition Graph (PPG) [10,14], our algorithm operates uniformly on predicated blocks (known as hyperblocks [20]) and SEME (Single-Entry Multiple-Exit) regions comprising basic blocks and hyperblocks.
- The result of applying our algorithm to a region is optimal: partially dead code in the resulting program cannot be removed without changing the branching structure of the program or potentially introducing non-existent predicate defining instructions, which impair some program executions.
- We have implemented our PDE algorithm in the ORC compiler [19]. We apply PDE just before its instruction scheduling pass since doing so can potentially reduce critical path lengths along frequently executed paths [5]. We present statistic evidence about the PDE opportunities in all the 12 SPECint2000 benchmarks despite that DCE has been applied several times earlier. We obtain moderate performance improvements in 8 out of 12 benchmarks on an Itanium machine at small compilation overheads.

Our running example is the CFG depicted in Figure 1, where the three regions formed are highlighted by dashed lines. Suppose the compiler applies if-conversion [1] to eliminate the branches in block 2. This leads to Figure 1(b), where the blocks 2 – 5 have been merged into a new hyperblock HB . In addition, we have inserted the so-called *interface blocks* [4], I_1 – I_4 at all region exits to simplify the design and implementation of our algorithm. Let us explain briefly the effectiveness of our algorithm using the SEME region R_2 depicted in Figure 1(b). To begin with, the assignment $x = b + d$ in block 6 is moved into block I_3 . As for the $y = c + d$ (p_3) in block HB , nothing needs to be done. At this stage, we obtain Figure 1(c). For the other three assignments, our algorithm first sinks $a = c + e$ into blocks I_2 and I_3 to get Figure 1(d). By sinking $a = c + e$, we are able to eliminate the partial deadness of $y = a + c$ (along the edge (2, 3) in Figure 1(a)). In actuality, $y = a + c$ is first removed from HB and the so-called *compensation code* $y = a + c$ (p_4) is inserted into HB just after the predicate defining instruction for p_4 . Figure 1(e) depicts the resulting program so far. Finally, $x = b + c$ in block 1 is partially dead along path 1 – HB – 6 – I_3 . The elimination of this partial deadness is illustrated in Figure 1(f): $x = b + c$ has been removed from block 1 and then inserted into blocks I_1 and I_2 .

The rest of this paper is organized as follows. Section 2 gives the background information about regions, predication, if-conversions and PPGs. Section 3 presents our PDE algorithm. Section 4 proves its correctness, termination and optimality. In Section 5, we discuss the implementation of our algorithm in ORC and present our performance results for the SPECint2000 benchmarks. Section 6 compares with the related work. Section 7 concludes the paper.

2 Preliminaries

Our algorithm applies to any SEME region comprising basic blocks and hyperblocks. However, this work is carried out in the context of ORC, which is a region-based compiler for the Intel’s Itanium Processor family. Section 2.1 defines the regions handled by our algorithm and discusses if-conversion and regional CFGs. Section 2.2 introduces PPGs and the queries used by our algorithm.

2.1 Regions

The ORC compiler [19] uses a region formation algorithm similar to interval analysis. As a result, the division of a CFG into regions puts a hierarchical structure on the CFG, called the *region tree*. There are four kinds of regions: (1) loop regions, (2) SEME regions, (3) MEME (Multiple-Entry Multiple-Exit) regions, and (4) improper regions (or irreducible regions). By default, ORC produces MEMEs only temporarily in an intermediate step and eventually converts every such an MEME into multiple SEMEs (with tail duplication [20] if necessary).

Consider the three regions formed in Figure 1(a). The leaf R_2 is an SEME consisting of blocks 1 – 6. The leaf R_3 is a loop region formed by block 7 alone. The root region R_1 contains R_2 , R_3 and block 8.

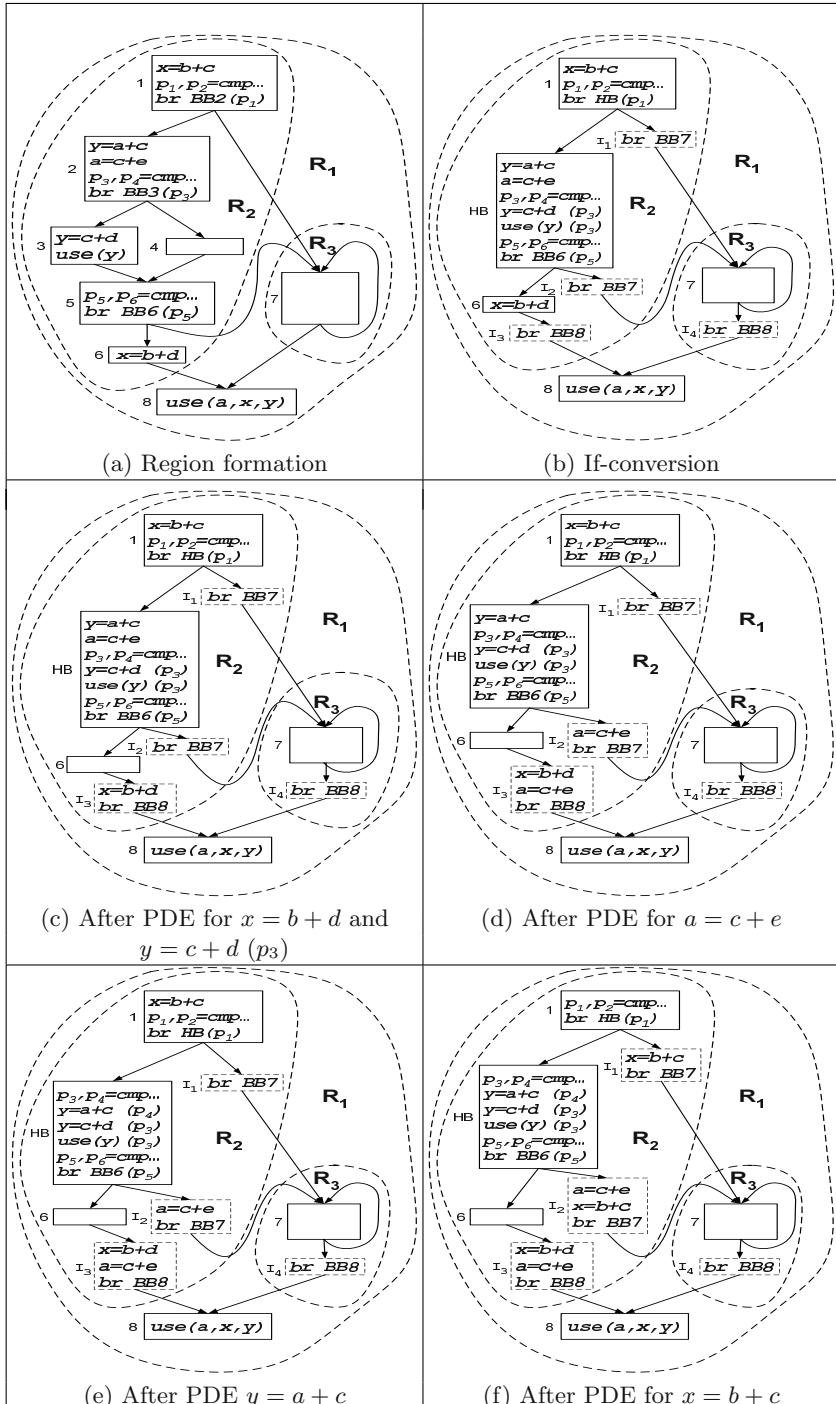


Fig. 1. A running example.

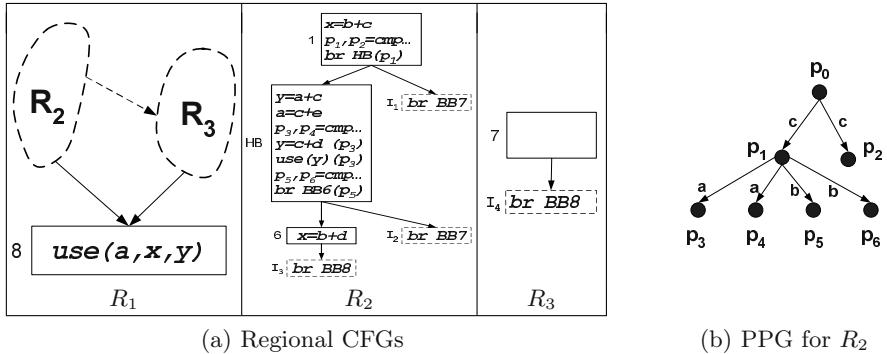


Fig. 2. Regional CFGs and the PPG for R_2 in our running example.

We assume that if-conversion is applied to the regions created during the region formation phase wherever appropriate. Often, an SEME subregion within a region is fully converted into a single, branch-free block of predicated code. The resulting block is known as a *hyperblock* [20].

Our algorithm operates on a region by traversing the nodes in its regional CFG. Figure 2(a) shows the regional CFGs for our example. As in the ORC compiler, the back edge around block 7 is not included in the regional CFG of the loop region R_3 . This is because back edges cannot be handled by if-conversion. Note that for each leaf region, we have inserted the interface blocks [4] (depicted in dashes), one for each of its region exits. As a result, the successors of every original node in a leaf region must all reside in that region.

In this study, we restrict ourselves to two kinds of leaf regions: SEMEs and innermost loop regions. So the innermost loop regions are considered as SEMEs. Therefore, our PDE algorithm is applicable to SEMEs only.

2.2 Predicate Partition Graph

Our algorithm eliminates partial deadness in an SEME region using the unique Predicate Partition Graph (PPG) [10,14] associated with the region. From now on, the term block means either a basic block or a hyperblock. The PPG tracks uniformly control flow and explicit use of predicates in a region. A predicate assigned to a block is called a *control predicate* and a predicate which explicitly appears in the instruction is called a *materialised predicate*. The control predicate of a block is viewed as a predicate combining all of the conditions which control whether the block will be executed or not. By convention, the control predicate of the unique entry block of an SEME region is p_0 , which denotes the always true predicate. Thus, every instruction can put into the following predicated form:

$$v = \pi \ (p)$$

where v is a variable, π an expression and p its materialised predicate. If $p = p_0$, we simply write the instruction as $v = \pi$ (without the materialised predicate).

Let α be an instruction in a block. The following notations are used:

- BB_α : the block in which α resides
- $M\text{-PRED}(\alpha)$: the materialised predicate of α
- $C\text{-PRED}(BB_\alpha)$: the control predicate of BB_α
- $E\text{-PRED}(\alpha)$: the *executing predicate* of α such that α is executed iff it holds:

$$E\text{-PRED}(\alpha) = \begin{cases} C\text{-PRED}(BB_\alpha) & \text{if } M\text{-PRED}(\alpha) = p_0 \\ M\text{-PRED}(\alpha) & \text{otherwise} \end{cases}$$

Consider the regional CFG for R_2 shown in Figure 2(a), which consists of three blocks 1, HB and 6. We find that $C\text{-PRED}(1) = p_0$, $C\text{-PRED}(HB) = p_1$ and $C\text{-PRED}(6) = p_5$. Consider the two instructions $a = c + e$ and $y = c + d$ (p_3) in HB . We find that $M\text{-PRED}(a = c + e) = p_0$ and $M\text{-PRED}(y = c + d (p_3)) = p_3$. Thus, $E\text{-PRED}(a = c + e) = C\text{-PRED}(HB) = p_1$ and $E\text{-PRED}(y = a + c (p_3)) = p_3$.

In a PPG, a node represents a predicate p and a directed edge (p, q) represents that there exists a partition r in p such that q is a subset of r [10,14]. Figure 2(b) depicts the PPG for the region R_2 in our example, where the edges from the same partition are conventionally decorated to have the same label. For example, p_1 has two distinct partitions: $p_1 = p_3 \cup p_4$ and $p_1 = p_5 \cup p_6$.

Our PDE algorithm relies on the following queries on a region's PPG. Our illustrating example is R_2 shown in Figure 2(a) and its PPG in Figure 2(b).

- $\text{IsDisjoint}(p, q)$: asks whether the domain of predicate p overlaps with that of predicate q . Two predicates are disjoint iff they can reach a common ancestor in the PPG through different edges of the *same partition*. For example, $\text{IsDisjoint}(p_3, p_4) = \text{IsDisjoint}(p_4, p_2) = \text{true}$ but $\text{IsDisjoint}(p_4, p_5) = \text{false}$.
- $\text{IsSubset}(p, q)$: asks whether the domain of p is a subset of the domain of q . For example, $\text{IsSubset}(p_3, p_1) = \text{IsSubset}(p_3, p_0) = \text{true}$ but $\text{IsSubset}(p_3, p_4) = \text{false}$. We shall write $p \subseteq q$ if $\text{IsSubset}(p, q)$ holds and $p \subset q$ if $p \subseteq q$ but $p \neq q$.
- $\text{LUB_Diff}(p, q)$: returns the set of predicates such that the union of their domains is the smallest superset of the domain of p subtracted by the domain of q , i.e., $\text{LUB_Diff}(p, q) \supseteq p - q$, where the equality holds when $q \subseteq p$ [10,14]. In our algorithm, $\text{LUB_Diff}(p, q)$ is called only when $q \subset p$. In addition, the result of this operation is simplified such that if all child predicates in a partition appear in $p - q$, then these child predicates are replaced with their parent predicate. For example, $\text{LUB_Diff}(p_1, p_3) = \{p_4\}$ and $\text{LUB_Diff}(p_0, p_3) = \{p_2, p_4\}$.

We assume that all the critical edges in the regional CFG of a region have been split. This simplifies the construction of the PPG for the region and the code insertions required in code motion/sinking transformations.

3 Region-Based PDE on Predicated Code

There are two main challenges in designing a PDE algorithm that works for regions comprising both basic blocks and hyperblocks. First, we must handle uniformly explicit branches and if-converted branches. We solve this first problem

by using a region’s PPG to guide the PDE process. Second, sinking an instruction across a branching node and later a join node is not straightforward in a worklist solution. Once again the branches at these branching and join nodes can be explicit or if-converted branches. We solve this second problem by sinking copies of an instruction with appropriate predicates at a branching node. We use a forest as a data structure to record the arriving copies at a join node. We combine the arriving copies at the join node into a single instruction once we have detected that a copy has arrived from each of its incoming edges by comparing predicate relations. Section 3.1 introduces our criterion of optimality. Section 3.2 presents our PDE algorithm. Section 3.3 illustrates it with our running example.

3.1 Scope

Our PDE algorithm operates on SEME regions, one at a time. It achieves a complete removal of partial deadness in an SEME region in the following sense.

Definition 1 (Optimality). *Let α be any instruction in an SEME region R , which is regarded as being distinct from every other instruction in R . After having completed assignment sinking and elimination for all instructions in R , let $\alpha_1, \dots, \alpha_n$ be all the copies of α in the resulting program, which satisfy:*

1. $E\text{-PRED}(\alpha_1) \subseteq E\text{-PRED}(\alpha), \dots, E\text{-PRED}(\alpha_n) \subseteq E\text{-PRED}(\alpha)$,
2. $E\text{-PRED}(\alpha_1), \dots, E\text{-PRED}(\alpha_n)$ are existing predicates in the PPG of R , and
3. $E\text{-PRED}(\alpha_1), \dots, E\text{-PRED}(\alpha_n)$ are mutually disjoint.

Such a transformation, which is assumed to be semantics-preserving, is optimal if $\cup_{i=1}^n E\text{-PRED}(\alpha_i)$ is the smallest possible.

Our PDE algorithm guarantees this optimality. Since we apply PDE before instruction scheduling, all necessary optimisations have already been performed. Therefore, we refrain from changing the branching structure of the program. We use only the existing predicates that are subsets of $E\text{-PRED}(\alpha)$. Thus, we do not introduce any new predicate defining instructions. Implicit in the optimality criterion is that the dynamic count of instructions along any path is not increased.

In comparison with Knoop, Rüthing and Steffen’s PDE algorithm on non-predicated code [18], our algorithm shares the same two properties as illustrated in Figures 3(a) and (b). First, we eliminate the partial deadness shown in Figure 3(a) by performing assignment sinking. In the transformed code, the executing predicate $p - q$ for the instruction in block 3 satisfies $p - q \subseteq p$. Second, we do not eliminate the partial deadness shown in Figure 3(b) since p and q are not related. However, we can remove the partial deadness of $x = a + b$ (p) along path 1-3-4 in two ways. If control flow restructuring is used as in [5], it is possible to ensure that the dynamic count of instructions is not increased along any path. But the new predicates introduced due to restructuring may increase the pressure for predicate registers. If restructuring is not used, some new predicate defining instructions may be introduced along some path. As a

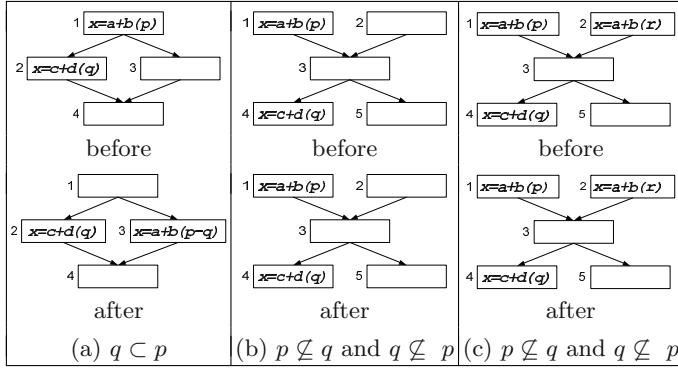


Fig. 3. Scope of our PDE algorithm.

result, the dynamic count of instructions along the path will be increased. This explains the existence of Restrictions 1 and 2 in Definition 1. Unlike [18], however, Figure 3(c) shows that we do not remove the partial deadness removable by simultaneously sinking two distinct instructions that happen to be identical. We can modify our algorithm slightly to deal with this scenario but have not done so since this case should not occur frequently in real code. This is why in Definition 1 all instructions are regarded as being distinct. Finally, Restriction 3 in Definition 1 is a basic requirement of any PDE algorithm.

3.2 Algorithm

Both code motion and deadness are governed by data dependences. There are three kinds of dependences between two distinct instructions: $\alpha : v = \pi$ and δ :

- $\text{DEFINED}(v, \delta)$: v is modified by δ in BB_δ .
- $\text{USED}(v, \delta)$: v is used by δ in BB_δ .
- $\text{KILLED}(\pi, \delta)$: some operands of π are modified by δ in BB_δ .

which are used to define the two predicates used directly by our algorithm:

- $\text{DEP}(\alpha, \delta) =_{df} \text{DEFINED}(v, \delta) \vee \text{USED}(v, \delta) \vee \text{KILLED}(\pi, \delta)$. There is a data dependence between α and δ iff $\text{DEP}(\alpha, \delta)$ holds.
- $\text{DEFINED-NOT-USED}(\alpha, \delta) =_{df} \text{DEFINED}(v, \delta) \wedge \neg \text{USED}(v, \delta)$. Essentially, α is partially dead with respect to δ only if $\text{DEFINED-NOT-USED}(\alpha, \delta)$ holds.

Figure 4 gives our algorithm, called *PPDE*, for performing PDE on an SEME region consisting of basic blocks and hyperblocks. Line 2 creates the empty interface blocks at all region exits as illustrated for R_2 in Figure 1(b). This ensures that all successors of every original node in R are contained in R itself. This simplifies the design of our algorithm so that we can move code out of R easily (line 21). Line 3 initialises W with all PDE candidates sorted in the reverse

```

1 PROCEDURE PPDE ( $R$ : SEME Region)
2 Create an empty interface block (with a branching instruction) for each region exit
3 Initialise the worklist  $W$  with all PDE candidates in  $R$ 
   sorted in the reverse topological order of their data dependences
4 while  $W$  is not empty
5    $\alpha$  = first PDE candidate from  $W$ , which, say, has the form  $v = \pi(p)$ 
6   Remove  $\alpha$  from  $W$ 
7   if  $v$  is not live out of  $R$  and the def-use chain of  $v$  in  $R$  is empty
8     Delete  $\alpha$  from  $BB_\alpha$            //  $\alpha$  fully dead
9     continue
10  if  $\alpha$  does not have the same form  $v = \pi$  as  $Prev$ 
11     $P = NULL$ 
12     $Prev = \alpha$ 
13   $\beta$  = instruction next to  $\alpha$  in  $BB_\alpha$     //  $\beta = NULL$  if  $\alpha$  is the last in  $BB_\alpha$ 
14  if ( $\neg Sink(R, \alpha, \beta)$ )
15    for each descendant block  $BB$  of  $BB_\alpha$  in  $R$  sorted in topological order
16      if  $\neg IsDisjoint(E-PRED(\alpha), C-PRED(BB))$ 
17         $\beta$  = first instruction of  $BB$ 
18        if  $Sink(R, \alpha, \beta)$ 
19          break
20  Delete the empty blocks (including empty interface blocks)
21  Move the interface blocks at the non-main exits of  $R$  into the parent region of  $R$ 

```

Fig. 4. The PDE algorithm on predicated code.

topological order of their dependences. *PPDE* terminates when W is empty (line 4). The PDE candidates in the worklist are processed sequentially, one at a time. As the algorithm proceeds, the candidates are removed from and new candidates added only to the beginning of the worklist in lines 43, 71 and 79. Thus, it is sufficient to understand how one PDE candidate is processed.

In lines 5 – 6, we remove the first PDE candidate α from the worklist. Like the existing PDE algorithms [5,11,18], we make use of live-in/out and def-use chains and assume that this information is updated wherever appropriate. Hence, in lines 7 – 8, we delete α when it is fully dead and are done with this instruction. Otherwise, line 14 calls *Sink* to perform sinking and elimination in the block containing α . If this returns false, we continue the PDE process for α in the descendant blocks BB rooted at BB_α in region R . We skip every descendant block that is not on a path starting from α since α cannot be partially dead there (line 16). The PDE process for α is completed when the call to *Sink* in line 18 returns true or when all such descendant blocks have been processed. Then the same PDE process repeats on the next candidate in the worklist.

Lines 10 – 12 are concerned with sinking the multiple copies of an instruction created at a branching node. This will be explained in Section 3.3.

At the end of *PPDE*, we do two things. In line 20, we clear up the regional CFG by deleting all empty blocks such as block 6 in Figure 1(c). In line 21, we move the interface blocks at the non-main region exits into the parent region of R . This strategy tends to reduce the critical path lengths along the frequently executed paths leaving the main exit(s) of the region R [5].

```

22 PROCEDURE Sink( $R$ : SEME Region,  $\alpha, \beta$ : Instruction)
23  $p = E\text{-PRED}(\alpha)$ 
24 for ( $\delta = \beta; \delta \neq NULL; \delta$  = instruction next to  $\delta$  in instruction list of  $BB_\beta$ )
25    $q = E\text{-PRED}(\delta)$ 
26   if  $\neg \text{IsDisjoint}(p, q)$ 
27   Case 1 if  $p = q$ 
28     if  $\text{DEP}(\alpha, \delta)$ 
29       if  $\text{DEFINED-NOT-USED}(\alpha, \delta)$ 
30         Delete  $\alpha$  from  $BB_\alpha$            //  $\alpha$  is fully dead
31         return true
32       else
33         if  $\delta$  is an exit of  $BB_\beta$ 
34           Move  $\alpha$  before  $\delta$ 
35           return false
36         else
37           Move  $\alpha$  after  $\delta$  and then set  $\delta$  to point to  $\alpha$ 
38   Case 2 else if  $p \supset q$ 
39     if  $\neg \text{DEFINED-NOT-USED}(\alpha, \delta)$ 
40        $\theta = \text{Create a copy of } \alpha$ 
41        $C\text{-PRED}(\theta) = C\text{-PRED}(\delta), M\text{-PRED}(\theta) = M\text{-PRED}(\delta)$ 
42       Insert  $\theta$  before  $\delta$ 
43       Insert  $\theta$  at the beginning of  $W$ 
44     else do nothing           //  $\alpha$  is partially dead
45     CompensationInsert( $R, \alpha, \delta$ )
46     Delete  $\alpha$  from  $BB_\alpha$ 
47     return true
48   Case 3 else if  $p \subset q$ 
49     if  $\text{DEP}(\alpha, \delta)$ 
50       if  $\text{DEFINED-NOT-USED}(\alpha, \delta)$ 
51         Delete  $\alpha$  from  $BB_\alpha$            //  $\alpha$  is fully dead
52         DelInst( $P, \alpha$ )
53         return true
54       else
55         if  $\text{AddInst}(P, \alpha, \delta)$ 
56         return true
57   Case 4 else
58     if  $\text{DEP}(\alpha, \delta)$ 
59     return true
60 return false
61 PROCEDURE DelInst( $P$ : Forest,  $\alpha$ : Instruction)
62 Delete the node  $\alpha$  and its parent if  $\alpha$  is its unique child from  $P$  (if  $\alpha$  exists)
63 PROCEDURE AddInst( $P$ : Forest,  $\alpha, \delta$ : Instruction)
64 if the node  $\alpha$  does not exist in  $P$ 
65   Add the directed edge  $\delta \rightarrow \alpha$  to  $P$ 
66   Let  $\alpha_1, \dots, \alpha_n$  be all children of  $\delta$  in  $P$ 
67   if  $\bigcup_{i=1}^n E\text{-PRED}(\alpha_i) = E\text{-PRED}(\delta)$ 
68      $\theta = \text{Create a copy of } \alpha$ 
69      $C\text{-PRED}(\theta) = C\text{-PRED}(\delta), M\text{-PRED}(\theta) = M\text{-PRED}(\delta)$ 
70     Insert  $\theta$  before  $\delta$ 
71     Insert  $\theta$  at the beginning of  $W$ 
72     Delete the nodes  $\alpha_1, \dots, \alpha_n$  and  $\delta$  from  $P$ 
73     Delete  $\alpha_1, \dots, \alpha_n$  from  $BB_{\alpha_1}, \dots, BB_{\alpha_n}$ 
74     return true
75 return false

```

Fig. 4. The PDE algorithm on predicated code (cont'd).

```

76 PROCEDURE CompensationInsert( $R$ : SEME Region,  $\alpha, \delta$ : Instruction)
77 for each predicate  $r$  in LUB_Diff(E-PRED( $\alpha$ ), E-PRED( $\delta$ ))
78    $\theta$  = Create a copy of  $\alpha$  such that E-PRED( $\delta$ ) =  $r$ 
79   Insert  $\theta$  at the beginning of  $W$ 
80    $BB$  = the block in which  $r$  is defined
81   if  $BB = BB_\delta$  //  $BB_\delta$  is a hyperblock
82     Insert  $\theta$  after the predicate defining instruction for  $r$ 
83   else
84     for each successor  $BB_s$  of  $BB$  in the region CFG of  $R$ 
85       if C-PRED( $BB_s$ ) =  $r$ 
86         Insert  $\theta$  at the entry of  $BB_s$ 
87         break

```

Fig. 4. The PDE algorithm on predicated code (cont'd).

3.2.1 Sinking and Elimination

The procedure *Sink* aims at eliminating the partial deadness of α in BB_α with respect to the instruction δ starting from β in BB_β . In lines 23 and 25, p and q are the executing predicates of α and δ , respectively. The procedure is driven entirely by comparing the predicate relations between p and q . If $\text{IsDisjoint}(p, q) = \text{true}$ in line 26, the executions of α and δ are mutually exclusive. Thus, α is not partially dead with respect to δ . The next iteration of **for** loop in line 24 is executed. Otherwise, there are the four cases depending on the relations between p and q . In Case 1, lines 33 – 37 have the effect of moving α into BB_δ when α and δ are in two different blocks. Case 2 is concerned with a branching point while Case 3 a merging point. In Case 4, α cannot be sunk any further if there is a dependence from α to δ .

3.2.2 Compensation Code Insertion

The procedure *CompensationInsert* inserts copies of α at the indicated insertion points with the executing predicates in LUB_Diff(E-PRED(α), E-PRED(δ)).

3.3 Example

Let us trace briefly the execution of our algorithm on the region R_2 shown in Figure 1(b). In line 2, the three interface blocks $I_1 - I_3$ are created as shown in Figure 1(b). In line 3, $W = (x = b+d, y = c+d (p_3), a = c+e, y = a+c, x = b+c)$, where the instructions have been sorted in the reverse topological order of their dependences. Consider the first PDE candidate $\alpha =_{df} x = b+d$ in block 6, where x is live out of R_2 . Since $\beta = \text{NULL}$ in line 13, the call to *Sink* in line 14 returns false immediately. I_3 is the unique successor of block 6. In line 17, β is set to point to the singleton instruction $brBB8$ in block I_3 . During the call to *Sink* in line 18, Case 1 is executed so that $x = b+d$ will be moved into the beginning of block I_3 . This is all that can be done for the first PDE candidate. Let us see how *PPDE* deals with the second PDE candidate $\alpha =_{df} y = (c+d) (p_3)$ in W . This time, $\beta =_{df} \text{use}(y) (p_3)$ in line 13. *Sink* is called in line 14. Case 1 is executed

again. Due to the flow dependence between the two instructions, *Sink* simply returns true in line 31. The resulting program so far is depicted in Figure 1(c).

At this time, $W = (a = c + e, y = a + c, x = b + c)$. *PPDE* removes $\alpha =_{df} a = c + e$ from W in lines 5 – 6 and sets $\beta =_{df} p_3, p_4 = cmp\dots$ in line 13. After lines 10 – 12, we have $P = NULL$ and $Prev =_{df} a = c + e$. In line 14, *Sink* is called. In the first iteration of the **for** loop in line 24, Case 1 is executed since $p = q = p_1$. So $a = c + e$ and $p_3, p_4 = cmp\dots$ are swapped. In the next iteration, δ points to $y = c + d$ (p_3). Hence, $q = p_3$. Since $p_1 \supset p_3$, Case 2 is executed. Note that $\text{DEFINED-NOT-USED}(\alpha, \delta) = \text{false}$. Thus, $a = c + e$ is removed from HB and $a = c + e$ (p_4) and $a = c + e$ (p_3) are inserted after $p_3, p_4 = cmp\dots$ in that order. Both instructions are added to W so that $W = (a = c + e (p_4), a = c + e (p_3), y = a + c, x = b + c)$. Finally, *Sink* returns true to *PPDE*.

Next, *PPDE* removes $\alpha =_{df} a = c + e$ (p_4) from W . As before, $P = NULL$ and $Prev =_{df} a = c + e$. In line 14, *Sink* is called. Eventually, α is moved just before $\delta =_{df} p_5, p_6 = cmp\dots$ in Case 1. Since $p = p_4$ and $q = p_1$, we have $p_4 \subset p_1$. Then Case 3 is executed, during which *AddInst* is called. The edge set for the forest P becomes $\{p_5, p_6 = cmp\dots \rightarrow a = c + e (p_4)\}$. *AddInst* returns false. The remaining PDE process for $a = c + e$ (p_4) does not cause any code motion. When *PPDE* removes $\alpha =_{df} a = c + e$ (p_3) from W , P remains unchanged after lines 10 – 12. In line 14, *Sink* is called. Eventually, δ will point to $p_5, p_6 = cmp\dots$ in Case 1. Then $q = p_1$. Since $p_3 \subset p_1$, Case 3 is executed, during which *AddInst* is called again. In line 65, the edge set for P becomes $\{p_5, p_6 = cmp\dots \rightarrow a = c + e (p_4), p_5, p_6 = cmp\dots \rightarrow a = c + e (p_3)\}$, where $p_3 \cup p_4 = p_1$. In the **if** statement beginning in line 67, $a = c + e$ (p_4) and $a = c + e$ (p_3) are combined and the resulting instruction $a = c + e$ is inserted just before $p_5, p_6 = cmp\dots$ and into W . Thus, $W = (a = c + e, y = a + c, x = b + c)$. The processing of $a = c + e$ will cause it to be split in Case 2 of *Sink* into two copies that are eventually moved into blocks I_2 and I_3 as shown in Figure 1(d).

At this time, $W = (y = a + c, x = b + c)$. By applying *PPDE* to $y = a + c$, Case 2 of *Sink* is executed. The partial deadness of this instruction is eliminated as shown in Figure 1(e). The result of performing PDE on the last candidate $x = b + c$ in block 1 is shown in Figure 1(f). Basically, $x = b + c$ is split at the end of block 1 such that one copy is moved into I_1 and the other, which is moved into HB , is dealt with in the same way as $a = c + e$ before. The only difference is that $x = b + c$, which is partially dead along path $1 - HB - 6 - I_3$, is eliminated.

Finally, all three interface blocks $I_1 - I_3$ are not empty. Those at the non-main region exits are moved into the parent region of R_2 , i.e., R_1 .

4 Correctness, Termination, Optimality

Theorem 1 (Correctness). *PPDE* preserves the semantics of the program.

Proof. We argue that every assignment sinking or elimination preserves the semantics of the program. In line 8, we delete α because it is fully dead. Let us examine Cases 1 – 3 in *Sink*. In Case 1, we delete α because it is fully dead (line



Fig. 5. The code generation (CG) module in ORC with PDE incorporated.

30). The **else** statement is justified due to the lack of dependences between α and δ . In Case 2, we delete α because it is partially dead (line 46). In addition, the compensation instructions are inserted correctly by *CompensationInsert* (line 45). Otherwise, α is not partially dead. But we have also inserted a copy of α , called θ , correctly in lines 40 – 42. In Case 3, we delete α because it is fully dead (line 51), and also update the data structure P (line 52). In the **else** statement, we combine the instances of α arriving at a merging point only when the equality in line 67 holds. The semantics of the program is preserved by lines 68 – 74.

Theorem 2 (Termination). *PPDE terminates.*

Proof. We argue that W , which starts with a finite number of PDE candidates, will be empty. Let α_i and α_{i+1} be two adjacent candidates in W . After having removed α_i from W , *PPDE* will eventually remove α_{i+1} from W . To see this, we note that during the PDE process on α_i , *Sink* can add new PDE candidates only at the beginning of W . There can only be finitely many since R is cycle-free. \square

Theorem 3 (Optimality). *PPDE is optimal.*

Proof. As a loop invariant for the **while** loop in line 4, all the PDE candidates in W are always in the reverse topological order of their data dependences. Each PDE candidate is moved downwards as far as possible: some of its copies are deleted iff they are dead, some are blocked due to dependences and the remaining ones are moved into interface blocks. So *PPDE* is optimal by Definition 1.

5 Experimental Results

We have implemented our PDE algorithm in the code generation (CG) module of the Open Research Compiler (ORC) [19] (version 2.1). Figure 5 depicts the compiler framework in which our PDE algorithm is used and evaluated. Our PDE pass is invoked just before the instruction scheduling pass. This phase-ordering not only eliminates all partial deadness before scheduling (Theorem 3) but also achieves an overall effect of reducing the cycles distributed into the eight Itanium stall categories (Figure 7).

We evaluate this work using all 12 SPECint2000 benchmarks. The benchmarks are compiled at the “O2” optimisation level with inlining switched on (except for **eon**). Inlining enables the frequently executed blocks in multiple functions to be formed into a single region. The profiling information is collected using the train inputs and is used by ORC in all profile-guided optimisations (including region formation). However, all benchmarks are executed using the reference inputs. The measurements were performed on an Itanium machine

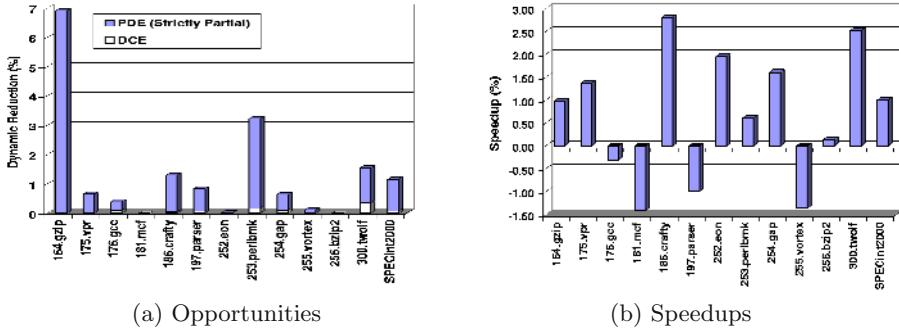


Fig. 6. Opportunities and speedups for SPECint2000.

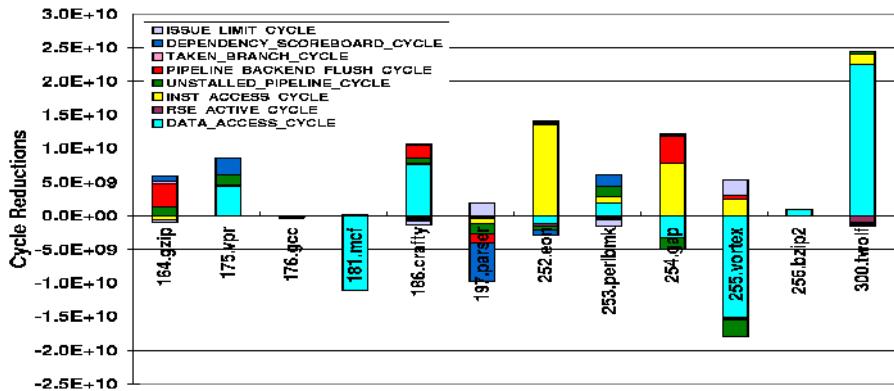


Fig. 7. Cycle reductions in stall categories for SPECint2000.

equipped with a 667MHz Itanium processor and 1GB of memory. We report the PDE opportunities, performance speedups and compilation overheads for the benchmarks. We also collect dynamic execution statistics to understand how PDE impacts various cycle metrics for a program using the pfmon tool.

The PDE candidates are rather comprehensive, including instructions on logical operations, arithmetic operations, shift operations, move operations (between registers), float conversion operations (e.g., `fcvt`), zero-extension operations (e.g., `zxt`) and multimedia operations. The non-PDE candidates are typically those with side effects, including instructions on memory operations (e.g., load and store) and cache operations. Another reason for excluding load instructions is that the instruction scheduler in ORC tends to move them up against the control flow. The other non-PDE candidates are compare and branch instructions and any instructions marked as being non-movable by the ORC compiler.

Figure 6 shows the benefits of PDE for the benchmarks. In Figure 6(a), we see convincingly the existence of PDE opportunities. This is true even though ORC has applied DCE several times earlier in the compilation process. For each

program, we measure the opportunity as the dynamic count of partially dead instructions eliminated over the dynamic count of PDE candidates executed using the profiling information from the reference inputs. In Figure 6(b), we see the execution time speedups over the ORC configuration. The positive speedups are obtained in 8 out of 12 benchmarks. The speedups for `crafty` and `twolf` are 2.81% and 2.53%, respectively. The performance degradations for `gcc`, `mcf`, `parser` and `vortex` are noted. The overall speedup for SPECint2000 is 1.01%.

The implementation of our PDE algorithm accounts for small compilation overheads for all the benchmarks, which range between 0.41% and 3.40% with an average of 1.41%. The benchmarks are cross-compiled on a 2.6GHz Pentium 4 PC with 2GB memory running Redhat Linux 8.0. There are several reasons for this efficiency. First, the leaf regions are small. Second, the average number of join nodes in the leaf regions is also small. Third, the PDE candidates are processed in the reverse topological order of their dependences. So *PPDE* on a leaf region terminates quickly.

To understand how PDE affects performance, we use `pfmon` to measure dynamic execution statistics through the eight Itanium performance monitors. Figure 7 presents the cycle reductions in the eight Itanium stall categories [13] for each program in both the ORC and ORC+PDE configurations. The dominating category (i.e., the one with the largest cycle reduction in absolute value) is `PIPELINE_BACKEND_FLUSH_CYCLE` for `gzip`, `DEPENDENCY_SCOREBOARD_CYCLE` for `parser` and `perlmark`, `INST_ACCESS_CYCLE` for `eon` and `gap` and `DATA_ACCESS_CYCLE` for the remaining seven benchmarks. Clearly, PDE affects the cycles in the stall category `DATA_ACCESS_CYCLE` more profoundly than the other seven categories. This category counts the number of cycles that the pipeline is stalled when instructions are waiting for the source operands from the memory subsystem. Of the seven benchmarks for which `DATA_ACCESS_CYCLE` is the dominating stall category, the cycles in `DATA_ACCESS_CYCLE` are decreased in `vpr`, `gcc`, `crafty` and `twolf` but increased in `gcc`, `mcf` and `vortex`. This phenomenon may be attributed to the aggressive nature of code sinking inherent in our PDE algorithm. By sinking instructions as low as possible along the control flow, the lifetimes are decreased for some variables but increased for the others.

6 Related Work

Most existing PRE algorithms [6,7,15,17] and PDE algorithms [5,11,18] are developed for non-predicated code. They are inadequate when instructions are predicated. There are some earlier research efforts on performing PRE on predicated code [9,16]. In particular, Knoop, Collard and Ju's PRE algorithm is based on SI-graphs, which are not as widespread as hyperblocks. By avoid introducing new predicate defining instructions, their algorithm guarantees that the dynamic count of instructions along any path is not increased. Later, Collard and Djelic [9] introduce a PRE algorithm on a single hyperblock by using first-order logical operations on predicates. They allow the synthesis of new predicate defining instructions. As a result, the instruction count along some path can be impaired.

August [3] discusses by an example how to perform PDE for a single hyperblock based on a predicate flow graph (PFG) [2]. The IMPACT compiler [8] supports DCE on predicated code. The ORC compiler performs DCE only on non-predicated IRs. We are not aware of any earlier region-based PDE algorithm on predicated code that works uniformly on basic blocks and hyperblocks.

Several approaches to predicate analysis have been described in the literature [10,14,21]. The predicate query system (PQS) introduced in [10,14] is based on the PPG. This is the system implemented in the ORC compiler. PQS can accurately represent predication conforming the style of if-conversion. The Predicate Analysis System (PAS) introduced in [21] is more powerful since it can accurately accommodate arbitrary predicate formulations. Our algorithm can be easily adapted when a PAS-based system is used provided it also supports the queries on control and materialised predicates at the same time.

7 Conclusion

Region-based compilation increases scheduling opportunities, which is critical for improving the performance of programs running on ILP architectures. Predicated execution on these architectures is an effective technique for dealing with conditional branches. The contribution of this research is the development of a practical algorithm for performing region-based PDE on predicated code. This algorithm is optimal in the sense that it can eliminate all partial deadness that can be removed without changing the branching structure of the program or potentially introducing new predicate defining instructions. We have implemented this algorithm in the ORC compiler for the Intel’s Itanium Processor family. In our implementation, PDE is applied just before instruction scheduling. This strategy not only eliminates all partial deadness but also achieves an overall effect of reducing the cycles distributed into the eight Itanium stall categories. We present statistic evidence about the PDE opportunities in real code. We demonstrate that our PDE algorithm can achieve moderate performance improvements for the SPECint2000 benchmarks at small compilation overheads.

References

1. J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189. ACM Press, 1983.
2. David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication. In *30th ACM/IEEE International Symposium on Microarchitecture*, pages 92–103. IEEE Computer Society, 1997.
3. David Isaac August. *Systematic Compilation For Predicated Execution*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.
4. J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront scheduling: path based data representation and scheduling of subgraphs. In *32nd ACM/IEEE International Symposium on Microarchitecture*, pages 262–271, 1999.

5. Rastislav Bodik and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *ACM SIGPLAN' 97 Conference on Programming Language Design and Implementation*, pages 159–170. ACM Press, 1997.
6. Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant computations. In *ACM SIGPLAN' 98 Conference on Programming Language Design and Implementation*, pages 1–14, 1998.
7. Qiong Cai and Jingling Xue. Optimal and efficient speculation-based partial redundancy elimination. In *1st IEEE/ACM International Symposium on Code Generation and Optimization*, pages 91–102, 2003.
8. P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and Wen-Mei Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *18th International Symposium on Computer Architecture (ISCA)*, pages 266–275, New York, NY, 1991. ACM Press.
9. Jean-Francois Collard and Ivan Djelic. A practical framework for redundancy elimination on EPIC processors. Technical report, PRISM, 2000.
10. David M. Gillies, Dz ching Roy Ju, Richard Johnson, and Michael Schlansker. Global predicate analysis and its application to register allocation. In *29th ACM/IEEE International Symposium on Microarchitecture*, pages 114–125, 1996.
11. Rajiv Gupta, David A. Berson, and Jesse Fang. Path profile guided partial dead code elimination using predication. In *5th International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113, 1997.
12. Richard E. Hank, Wen-Mei Hwu, and B. Ramakrishna Rau. Region-based compilation: an introduction and motivation. In *28th ACM/IEEE International Symposium on Microarchitecture*, pages 158–168. IEEE Computer Society Press, 1995.
13. Intel. Intel Itanium processor reference manual for software development, December 2001.
14. Richard Johnson and Michael Schlansker. Analysis techniques for predicated code. In *29th ACM/IEEE International Symposium on Microarchitecture*, pages 100–113, 1996.
15. Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, and Peng Tu. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.
16. Jens Knoop, Jean-Francois Collard, and Roy Dz ching Ju. Partial redundancy elimination on predicated code. In *7th International Static Analysis Symposium*. Springer-Verlag, 2000.
17. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.
18. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *ACM SIGPLAN' 94 Conference on Programming Language Design and Implementation*, pages 147–158. ACM Press, 1994.
19. Yang Liu, Zhaoqing Zhang, Ruliang Qiao, and Roy Dz ching Ju. A region-based compilation infrastructure. In *7th Workshop on Interaction between Compilers and Computer Architectures*, 2003.
20. Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyper-block. In *25th ACM/IEEE International Symposium on Microarchitecture*, pages 45–54. IEEE Computer Society Press, 1992.
21. John W. Sias, Wen-Mei Hwu, and David I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *33rd ACM/IEEE International Symposium on Microarchitecture*, pages 112–123. ACM Press, 2000.

Value-Based Partial Redundancy Elimination

Thomas VanDrunen and Antony L. Hosking

Purdue University

Abstract. Partial redundancy elimination (PRE) is a program transformation that identifies and eliminates expressions that are redundant on at least one (but not necessarily all) execution paths. Global value numbering (GVN) is a program analysis and transformation that identifies operations that compute the same value and eliminates operations that are redundant. A weakness of PRE is that it traditionally considers only expressions that are lexically equivalent. A weakness of GVN is that it traditionally considers only operations that are *fully* redundant. In this paper, we examine the work that has been done on PRE and GVN and present a hybrid algorithm that combines the strengths of each. The contributions of this work are a framework for thinking about expressions and values without source-level lexical constraints, a system of data-flow equations for determining insertion points, and a practical algorithm for extending a simple hash-based GVN for PRE. Our implementation subsumes GVN statically and, on most benchmarks, in terms of performance.

1 Introduction

The goal of a compiler’s optimization phases is transforming the program to make it more efficient. Accordingly, researchers have given much attention to methods for eliminating redundant computations; both approaches and terminology to describe them have proliferated. In the progeny of this research, two primary genera can be identified (some cross-breeding notwithstanding): Partial Redundancy Elimination (PRE) and Global Value Numbering (GVN). The goal of the present work is to present an algorithm for a hybrid subsuming these two approaches synthesized from known techniques and tools.

1.1 Motivation

PRE considers the control flow of the program and identifies operations that are redundant on *some but not necessarily all* traces of the program that include that operation, and hence are *partially redundant*. It hoists operations to earlier program points where originally the operation was unknown and removes operations that the hoisting has rendered fully redundant. In Figure 1(a), $e \leftarrow c + b$ is partially redundant because of $d \leftarrow c + b$. By hoisting and preserving the result of the operation in temporary variable t , PRE produces the program in Figure 1(b). Because of this hoisting, PRE is sometimes called (an instance of) *code*

motion. GVN on the other hand considers the value produced by an operation (in this paper, we always assume *value* to mean static value; operations in loop bodies or otherwise reentrant code may result in different dynamic values at different points in program execution). If a value has already been computed, subsequent operations can be eliminated even though they may differ lexically. Because of the move $c \leftarrow a$ in Figure 1(a), the operations $c \leftarrow b + a$ and $d \leftarrow c + b$ compute the same value. Accordingly GVN preserves this value in a temporary and eliminates the re-computation, as in Figure 1(c). As this example shows, in their basic form, neither PRE nor GVN is strictly more powerful than the other (see Muchnick p. 343 [25]).

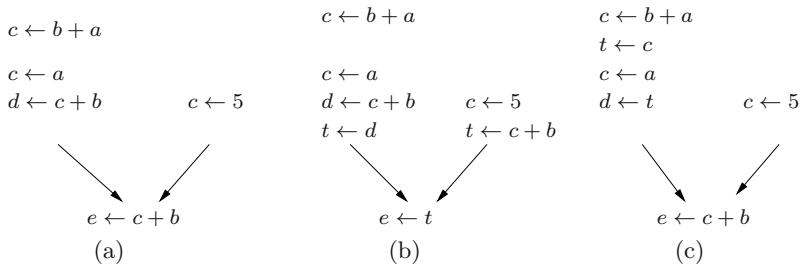


Fig. 1. Basic example

1.2 Our Result

Answering this challenge, we have found a technique, GVN-PRE, that subsumes both and eliminates redundancies that would be undetected by either working in isolation. We create a system of data flow equations that, using the infrastructure of a simple hash-based value numbering system, calculate insertion points for partially redundant values rather than partially redundant expressions. We present this in a framework that allows clear reasoning about expressions and values and implement it with an algorithm that can easily be reproduced. After exploring the background to this problem and related work, we explain the framework, give the data flow equations and related algorithm, and explore an implementation and its performance. Our implementation subsumes GVN.

1.3 Related Work

PRE. Partial Redundancy Elimination in general poses several challenges. First, lexically identical expressions at different program points do not necessarily compute the same result, since operands may be reassigned, so any PRE algorithm must take into account assignments that kill an expression's availability. Furthermore, care must be taken that the hoisting would do no harm: if an

operation is added at a program point, it must occur without reassignment to its operands on all paths from that point to program exit (in the literature, they say the operation must be *downsafe* or *anticipated* at that point). Otherwise the hoisting will lengthen at least one trace of the program, defying optimality; even worse, if the hoisted instruction throws an exception, the program's semantics change. Finally, PRE algorithms should resist the temptation to hoist operations earlier in the program than necessary; while this may not do any harm in terms of lengthening a trace of the program, it will make no improvement either, but may increase register pressure by lengthening the live range of an assignment.

PRE was invented by Morel and Renoise [24]. The original formulation used bit vectors to detect candidates for motion and elimination. Knoop, Rüthing, and Steffen gave a complete, formal, and provably optimal version of PRE which they called Lazy Code Motion (LCM) [19,20]; it was improved by Drechsler and Stadel [13]. LCM used a myriad of predicates and equation systems to determine the earliest and latest placement points for operations that should be hoisted. Chow, Kennedy et al. produced a PRE algorithm for programs in static single assignment form (SSA) [11], called SSAPRE [8,17]. SSA is an intermediate representation property such that program variables and temporaries are divided into different versions for each assignment so that each version of a variable is assigned to exactly once in a static view of the program. If a variable exists in different versions on incoming paths at a join point, they are merged into a new version at the join. SSA makes it easy to identify the live ranges of variable assignments and hence which lexically equivalent expressions are also semantically equivalent. On the other hand, it complicates hoisting since any operand defined by a merge of variable versions must have the earlier version back-substituted. More recently, Dhamdhere has presented a simpler PRE that examines specific paths from earlier to later occurrences of an operation [12].

A major handicap for these versions of PRE is that they are based on lexical (or syntactic) equivalences. Although SSA provides the freedom to think outside the lexical box because each assignment essentially becomes its own variable, even SSAPRE does not take advantage of this but regards expressions in terms of the source variables from which their operands come; in fact, it makes stronger assumptions about the namespace than basic SSA [34]. The fact that many redundant results in a program do not come from lexically identical expressions (nor do all lexically identical expressions produce the same results) has motivated research to make PRE more effective [5] and led Click to assert that “in practice, GVN finds a larger set of congruences than PRE” [9].

GVN. Global Value Numbering partitions expressions and variables into classes (or assigns them a unique *value number*) all of which have the same static value (or are *congruent*). A well-known form having its origin in two papers simultaneously presented by Alpern, Rosen, Wegman, and Zadek [3,28], it uses algebraic properties to associate expressions, such as assignments ($a \leftarrow b$ implies a and b are in the same class), commutativity of certain operations ($a + b$ is in the same class as $b + a$), and facts about certain constants ($a + 0$ is in the same class as a). SSA is a major boon for GVN, since (SSA versions of) variables are always in the

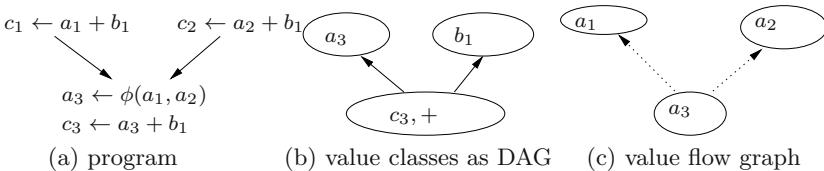


Fig. 2. Example showing weakness of traditional GVN and the use of a VFG

same class, and therefore so are expressions on those versions in the same class, statically and globally. Congruences can be found by hashing [9,6] or by first assuming all expressions to be equal and splitting classes when incongruences are found [3,16,30]. Since identifying an operation as redundant depends on the operation having the same *global* number as an earlier operation, in traditional GVN such an earlier operation must dominate (i.e., occur on all paths to) the later operation. However, if any given path is considered in isolation, some value classes can be merged. In Figure 2(a), $\phi(a_1, a_2)$ merges the variables a_1 and a_2 , so if we consider the left path, a_3 should be in the same class as a_1 , whereas on the right path, a_2 should be in the same class as a_1 . Yet the operation $a_3 + b_1$ will not be eliminated because nothing in its global class has been computed earlier. Since traditional GVN does no motion, it does not eliminate some operations that on any given trace of the program are fully redundant—let alone operations that are partially redundant. (Another tradition of global value numbering has been worked on by Steffen et al [31,32,33,29,22]. Its various versions have tended to be more general and theoretically complete but less amenable to implementation than that of Rosen, Alpern, et al.)

Hybrids. One of the original GVN papers [28] is remarkable because it not only introduces GVN, SSA, and critical edge removal; it also does PRE on values. However, the range of congruences it can identify is small, and subsequent research seems to have ignored this as it wallowed in its lexical mire. The first explicit study of the potential for PRE to work on values (or for GVN to consider partial redundancies) is that of Knoop, Rüthing, and Steffen’s comparison of code motion and code placement [21]. They distinguish *syntactic* (lexical) code motion from *semantic*, which considers values. They note that the collection of value classes can be modeled by a dag, since a value is represented by an expression (a node) which has other values as operands (edges pointing to nodes). The path-based connections among values is captured by what they call the *value flow graph* (VFG). See Figure 2(b) and (c). Knoop et al. use these constructs to sketch an algorithm for determining safe insertion points for code motion (PRE), but distinguish it from a more powerful optimization which they call code placement and for which determining insertion points traditionally considered safe is not adequate for finding all insertion points necessary for optimality.

Bodík and Anik, independently of Knoop et al., developed an algorithm that also addressed PRE on values [4]. Their work uses value numbering with back-

substitution to construct a *value name graph* (VNG) similar to Knoop et al.’s VFG. The VNG considers expressions to be names of values. If the program is not in SSA, expressions represent different values at different program points, and the nodes of the graph are expression / program-point pairs. The edges capture the flow of values from one name to another according to program control flow. Bodík et al. then use data flow analysis to identify optimal insertion points for PRE, an algorithm they call VNGPRE.

1.4 Overview

In this paper, we present a new algorithm for PRE on values which subsumes both traditional PRE and GVN. The contributions of our approach are as follows. First, our analysis considers larger program chunks than Bodík (basic blocks instead of single instructions). Second, we present a framework for expressions and values that takes full advantage of SSA, completely ignoring source-level lexical constraints and thinking solely in terms of values and expressions that represent them. Third, no graph is explicitly constructed nor is any novel structure introduced. Instead, we achieve the same results by synthesizing well-known tools (SSA, control flow graphs, value numbers) and techniques (flow equations, liveness analysis, fixed-point iteration). In fact, our algorithm can be viewed as an extension of a simple hash-based GVN. Finally, we report on an efficient and easily-reproduced implementation for this algorithm. Our intention here is not to make a head-on comparison with the approaches of Bodík et al. or Knoop et al., but rather to present an approach that covers both traditional PRE and GVN that is easily realized in implementation.

The rest of the paper is organized as follows: Section 2 gives preliminaries, such as assumptions we make about the input program, precise definitions of concepts we use for our analysis, and descriptions of global value numbering infrastructure we assume already available. Section 3 contains the meat of the paper, describing the various phases of the approach both formally and algorithmically. Section 4 gives details on our implementation and its results. We conclude by discussing future work in Section 5.

2 Framework

In this section, we present our framework in terms of program structure we assume and the model of expressions and values we use.

2.1 Program Structure

We assume the input programs are in an intermediate representation that uses a *control flow graph* (CFG) over *basic blocks* [1]. A basic block is a code segment that has no unconditional jump or conditional branch statements except for possibly the last statement, and none of its statements, except for possibly the first, is a target of any jump or branch statement. A CFG is a graph representation

of a procedure that has basic blocks for nodes and whose edges represent the possible execution paths determined by jump and branch statements. We define $\text{succ}(b)$ to be the set of successors to basic block b in the CFG, and similarly $\text{pred}(b)$ the set of predecessors. (When these sets contain only one element, we use this notation to stand for that element for convenience.) We also define $\text{dom}(b)$ to be the dominator of b , the nearest block that dominates b . This relationship can be modeled by a *dominator tree*, which we assume to be constructed [1]. We assume that all *critical edges*—edges from blocks with more than one successor to blocks with more than one predecessor [28]—have been removed from the CFG by inserting an empty block between the two blocks connected by the critical edge.

We assume the following language to define the set of possible instructions in the program (this excludes jumps and branches, which for our purposes are modeled by the graph itself):

$k ::= t \mid t \text{ op } s$	<i>Operations</i>
$p ::= \phi(t^*)$	<i>Phis</i>
$\gamma ::= k \mid p \mid \bullet$	<i>Right-hand terms</i>
$i ::= t \leftarrow \gamma$	<i>Instructions</i>
$b ::= i^*$	<i>Basic blocks</i>

The symbol \bullet stands for any operation which we are not considering for this optimization; it is considered to be a black box which produces a result. The meta-variable t ranges over (SSA) variables. We do not distinguish between source-level variables and compiler-generated temporaries, nor are we concerned with what source-level variable any SSA variable come from. Because of SSA form, no variable is reassigned, and a variable's scope is all program points dominated by its definition. For simplicity, we do not include constants in the grammar, though they appear in some examples; constants may be considered globally-defined temporaries. We let op range over operators.

A phi contains the same number of operands as there are predecessors to its block, and each operand is associated with a distinct predecessor. Semantically, the phi selects the operand associated with the predecessor that precedes the block on a given trace and returns that operand as its result. Phis precede all non-phis in a block.

2.2 Values and Value-Numbering

A *value* is a set of expressions. v ranges over values. An expression is similar to an operation except that if an expression involves an operator, its operands are given in terms of values rather than subexpressions. This way we can think of expressions more generally. If t_1 and t_3 are members of value v_1 , and t_2 is a member of v_2 , we need not think of $t_1 + t_2$ and $t_3 + t_2$ as separate expressions; instead, we think only of the expression $v_1 + v_2$.

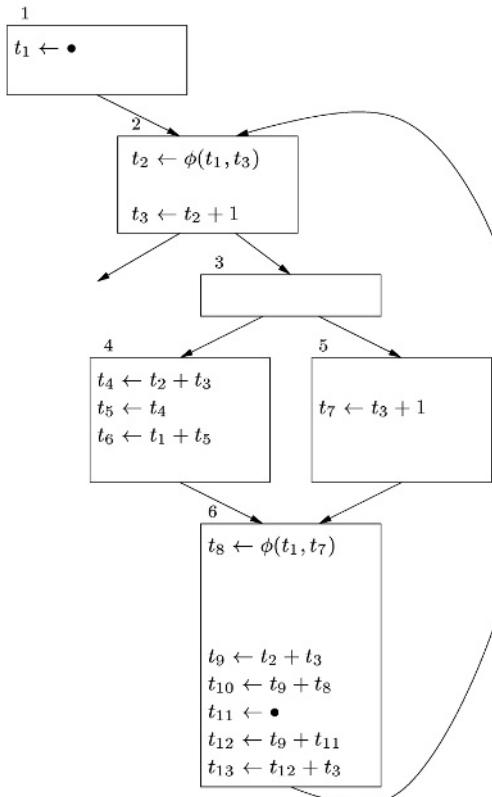
$$e ::= s \mid v \text{ op } v \quad \text{Expressions}$$

Our goal in value numbering is to obtain three pieces of information: value numbers, available expressions, and anticipated expressions. Traditional value numbering requires only the first two. First, we partition all expressions in the program into values. We represent this partition with a map (or value table) on which we can add and lookup the values for expressions. We treat the value table as a black box, but assume it to be used for a simple hashing value number scheme, recognizing expressions by their structure and, if smart enough, algebraic properties, associating them with values. It is worth noting that there is no limit on how smart the function `lookup` is engineered to be. For example, if v_1 contains 1, v_2 contains 2, v_4 contains $v_3 + v_1$ for some v_3 , and v_5 contains $v_4 + v_1$, then v_5 also should contain $v_3 + v_2$. To enumerate all such cases would be infinite.

Figure 3(a) displays a running example we will refer to throughout this and the next section. Since it is large, we will often discuss only part of it at a time. For the present, consider block 4. The table in Figure 3(b) displays the values referenced in the block and the expressions they contain. For clarity, we assign a value the same subscript as one of its temporaries wherever possible. Values v_1 , v_2 , and v_3 are defined in blocks that dominate block 4, so they are in scope here. The instruction $t_4 \leftarrow t_2 + t_3$ leads us to discover expressions $v_2 + v_3$ and t_4 as elements of the same value, which we call v_4 . Because of the move $t_5 \leftarrow t_4$, t_5 is also in v_4 . Finally, $v_1 + v_5$ and t_6 are in a value we call v_6 . Recall that these values are global, so we are showing only part of the value table for the entire program (in fact, values v_3 and v_4 have more members, discovered in other parts of the program).

The second piece of information is what expressions are available to represent the values; or, put differently, what values are already computed and stored in temporaries at a given program point. If more than one temporary of the same value is live at a point, we want to pick one as the leader, which will be used as the source of the value to replace an operation of that value with a move. To be unambiguous, the leader should be the “earliest” temporary available; that is, the temporary whose defining instruction dominates the defining instructions of all other temporaries of that value live at that point. We define the *leader set* to be the set of leaders representing available values at a program point. Although technically there is a leader set for any program point, for our purposes we will be interested only in the sets at the end of a basic block, `AVAIL_OUT`. The leaders available out of block 4 are those expressions listed first for each value in Figure 3(b). (This notion of availability is sometimes called *upsafety* [21].)

Finally, we want to know what values are *anticipated* at a program point; that is, it will be computed or used on all paths from that point to program exit. (This notion of anticipation—sometimes called *anticability*—is essentially the same as that of *downsafety*.) Just as we want appropriate temporaries to represent values in the leader set, so we want appropriate expressions to represent anticipated values—that is, anticipated values should also have a reverse-flow leader or *antileader*. An antileader can be a live temporary or a non-simple



(a) Program CFG

values

$v_1: t_1$
 $v_2: t_2$
 $v_3: t_3$
 $v_4: t_4, v_2 + v_3, t_5$
 $v_6: t_6, v_1 + v_4$

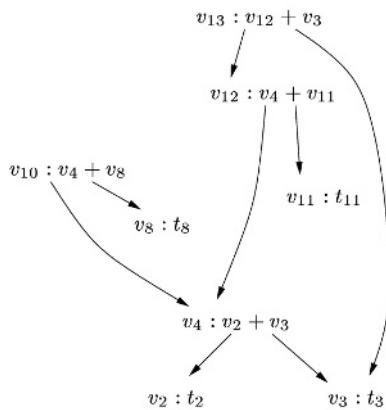
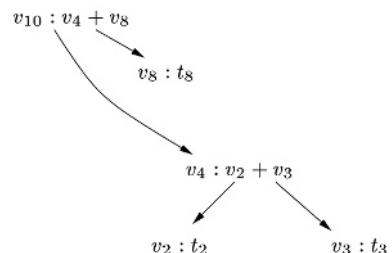
temps generated:

t_4, t_5, t_6

exprs generated:

$t_2, t_3, t_4, t_1, t_5, v_2 + v_3, v_1 + v_4$

(b) Values, temporaries, and expressions of block 4

(c) Anticipated expressions at block 6, not considering the kill of t_{11} 

(d) Actual anticipated expressions at block 6

Fig. 3. Running example

expression whose operands are represented in the antileader set. This is so that the expression could become available by an insertion at that point, or several insertions if it is a nested expression. Conversely from the leader set, we are interested only in the antileader sets at the beginning of basic blocks, ANTIC_IN.

For an antileader set, it does not matter which expression represents a value, so long as that value is live. A temporary potentially in ANTIC_IN becomes dead if it is assigned to. If the assignment is from something we can make an expression for (as opposed to \bullet), that expression replaces the temporary as the antileader. If the assignment is from \bullet , then the value is no longer represented at all. Furthermore, any other expression that has that (no longer represented) value as an operand also becomes dead. Therefore antileaders and the values they represent are killed by definitions of temporaries in the block. An antileader set can be pictured as a dag. Consider basic block 6 in Figure 3(a), alternately with and without the instruction $t_4 \leftarrow \bullet$. In the case where we exclude that instruction, assume t_4 to be global. Without the definition of t_4 , ANTIC_IN can be represented by the dag in Figure 3(c). The nodes of the dag are pairs of values and the antileaders representing them; edges are determined by the operands of the antileader. If we suppose the block contains $t_6 \leftarrow \bullet$ as it does in the program, then $v_4 : t_4$ is killed, along with all expressions that depend on v_4 also, in cascading effect. See Figure 3(d).

To calculate these sets, we use two other pieces of information: the temporaries generated by a set, meaning those that are defined; and the expressions generated by a set, meaning those that occur as operations, operands to operations, or sources of moves. We say that the set “generates” these because they are used as “gen” sets for the flow equations we define in the next section for the leader and antileader sets. These are given for block 4 in Figure 3(b).

3 GVN-PRE

The GVN-PRE algorithm has three steps: BuildSets, Insert, and Eliminate. BuildSets, which we describe formally and algorithmically, populates the value table and the leader and antileader sets. Insert places new instructions in the program to make partially available instructions fully available. Eliminate removes computations whose values are already available in temporaries or as constants.

3.1 BuildSets

Flow equations. To compute AVAIL_IN and AVAIL_OUT for a block, we must consider not only the contents of the block itself, but also expressions inherited from predecessors (for AVAIL_OUT) and anti-inherited from successors (for ANTIC_IN). For this we use a system of flow equations. As is common for flow equations, we also define the sets AVAIL_IN and ANTIC_OUT, although only AVAIL_OUT and ANTIC_IN are used later in the optimization. We have three gen sets, EXP_GEN(b) for expressions (temporaries and non-simple) that appear in the right hand side of an instruction in b ; PHI_GEN(b) for temporaries that

are defined by a phi in b ; and $\text{TMP_GEN}(b)$ for temporaries that are defined by non-phi instructions in b . There are no kill sets for calculating AVAIL_OUT , since SSA form implies no temporary is ever killed. For ANTIC_IN , TMP_GEN acts as a kill set. Since an available expression must be defined in an instruction that dominates the program point in question, so in calculating AVAIL_OUT , we consider inherited expressions only from the block's dominator. In terms of flow equations,

$$\text{AVAIL_IN}[b] = \text{AVAIL_OUT}[\text{dom}(b)] \quad (1)$$

$$\begin{aligned} \text{AVAIL_OUT}[b] = \text{canon}(\text{AVAIL_IN}[b] \cup \text{PHI_GEN}(b) \\ \cup \text{TMP_GEN}(b)) \end{aligned} \quad (2)$$

where canon is a procedure that, given a set of temporaries, partitions that set into subsets which all have the same value and chooses a leader from each. Of course canon would be inconvenient and inefficient to implement; instead, $\text{AVAIL_OUT}[b]$ can be calculated easily and efficiently at the same time as the gen sets, as we will show later.

For ANTIC_IN , handling successors is more complicated. If there is only one successor, we add all its antileaders to AVAIL_OUT of the current block; however, we must translate some temporaries based on the phis at the successor. For example, if t_1 is anticipated by block b , and block b has a phi which defines t_1 and has t_2 as an operand from block c , then t_2 , rather than t_1 , is anticipated at the end of block c , and it has a different value. For this we assume a function phi_translate which takes a successor block, a predecessor block (i.e., there is an edge from the second block to the first), and a temporary; if the temporary is defined by a phi at the successor, it returns the operand to that phi corresponding to the predecessor, otherwise returning the temporary. If there are multiple successors, then there can be no phis (because of critical edge removal), but only values anticipated by all of the successors can be anticipated by the current block. This assures that all anticipated expressions are downsafe—they can be inserted with assurance that they will be used on all paths to program exit.

The flow equations for calculating the antileader sets are

$$\text{ANTIC_OUT}[b] = \begin{cases} \{e | e \in \text{ANTIC_IN}[\text{succ}_0(b)] \wedge \\ \forall b' \in \text{succ}(b), \exists e' \in \text{ANTIC_IN}[b'] | \\ \text{lookup}(e) = \text{lookup}(e')\} & \text{if } |\text{succ}(b)| > 1 \\ \text{phi_translate}(A[\text{succ}(b)], b, \text{succ}(b)) & \text{if } |\text{succ}(b)| = 1 \end{cases} \quad (3)$$

$$\begin{aligned} \text{ANTIC_IN}[b] = \text{clean}(\text{canon}_e(\text{ANTIC_OUT}[b] \cup \text{EXP_GEN}[b] \\ - \text{TMP_GEN}(b))) \end{aligned} \quad (4)$$

When phi_translate translates expressions through phis, it may involve creating expressions that have not been assigned values yet and therefore require new values to be created. Consider calculating $\text{ANTIC_OUT}[B_5]$ in our example. Value v_{10} is represented by $v_4 + v_8$, which is translated through the phi to $v_4 + v_7$. However, that expression does not exist in the program—it needs a new value.

Thus sometimes the value table will need to be modified while calculating these sets.

`canone` generalizes `canon` for expressions. For `ANTIC_IN`, we do not care what expression represents a value as long as it is live, so `canone` can make any choice as long as it is consistent. This essentially makes the union in the formula for `ANTIC_IN` to be “value-wise”, meaning that given two sets of expressions representing values, we want every value in each set represented, but by only one expression each. Similarly, the formula for `ANTIC_OUT` when there are multiple successors performs a “value-wise” intersection—only values that are represented in all successors should be represented here, but which expression represents it does not matter. `clean` kills expressions that depend on values that have been (or should be) killed. Because information will also flow across back edges in the graph, these sets must be calculated using a fixed-point iteration.

Algorithm. Calculating BuildSets consists of two parts. The first is a top-down traversal of the dominator tree. At each block, we iterate forward over the instructions, making sure each expression has a value assigned to it. We also build `EXP_GEN`, `PHI_GEN`, and `TMP_GEN` for that block.

Computing `canon` directly is inconvenient and costly. We avoid computing it by maintaining an invariant on the relevant sets, that they never contain more than one expression for any value. Since the value leader set for the dominator will have already been determined, we can conveniently build the leader set for the current block by initializing it to the leader set of the dominator and, for each instruction, adding the target to the leader set *only if its value is not already represented*. This way, we never add a temporary to an `AVAIL_OUT` set unless its value is not yet represented. Similarly, we need only one representative in `EXP_GEN` for each value, so we do not add an expression for a value that has already appeared (this way `EXP_GEN` contains only the first appearance of a value, appropriate for an antileader). We do not calculate `AVAIL_IN` sets since it is trivial.

The second part calculates flow sets to determine the antileader sets and conducts the fixed-point iteration. Until we conduct a pass on which no `ANTIC_IN` set changes, we perform top-down traversals of the postdominator tree. This helps fast convergence since information flows backward over the CFG. To keep `ANTIC_IN` canonical, we remove the killed temporaries from `ANTIC_OUT` and `EXP_GEN` separately, and then do what amounts to a value-wise union.

Both `phi_translate` and `clean` process elements in such a way that other set elements that they depend on must be processed first. The key to doing this efficiently is to maintain all sets as topological sorts of the dags they model, which can be done by implementing the sets as linked lists. Figure 4 shows a round of this process on block 5, including `ANTIC_IN[B5]` before and after the insertions from S . Notice that the net effect is to replace $v_7 : t_7$ in `ANTIC_OUT` with $v_7 : v_3 + v_{100}$, and that although the order has changed, it is still a topological sort. For this to be efficient, the fixed point iteration must converge quickly; convergence depends primarily on CFG structure. In our benchmarks, the maximum number of iterations needed for any method being compiled was 55; that,

however, appears to represent an extreme outlier, as the second highest was 18. When measuring the maximum number of rounds needed for convergence of any method, 15 of our 20 benchmarks had a maximum less than 10, and we found that if we set 10 as the limit, there was no decrease in the number of operations eliminated in the elimination phase, implying that the extra rounds needed in extreme cases produce no useful data.

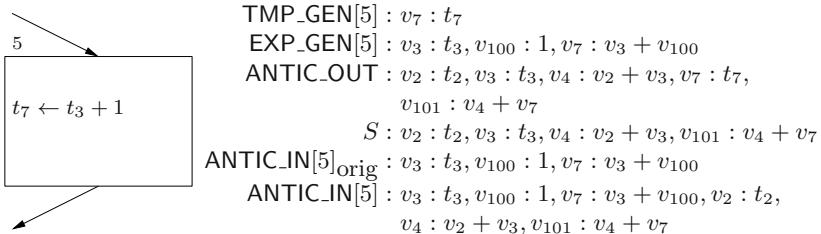


Fig. 4. Example for the second phase of BuildSets

3.2 Insert

Insert is concerned with hoisting expressions to earlier program points. If this phase were skipped, we would be left with a traditional global value numbering scheme. Insertions happen only at merge points. This phase iterates over blocks that have more than one predecessor and inspects all expressions anticipated there. For a non-simple expression, we consider the equivalent expressions in the predecessors. This requires some translation because of the phis at the block, for which we use `phi_translate`. We look up the value for this equivalent expression and find the leader. If there is a leader, then it is available. If the expression is available in at least one predecessor, then we insert it in predecessors where it is not available. Generating fresh temporaries, we perform the necessary insertions and create a phi to merge the predecessors' leaders.

We can see the results of Insert in Figure 5 at the join point of block 6. The anticipated expressions $v_2 + v_3$ and $v_4 + v_8$ are available from block 4, so $t_{16} \leftarrow t_2 + t_3$ and $t_{18} \leftarrow t_{16} + t_7$ are hoisted to block 5. Here the topological sort of $\text{ANTIC_IN}[B_6]$ comes in handy again, since these expressions are nested and $v_2 + v_3$ must be inserted first. Note that thus our algorithm handles *second order effects* without assigning ranks to expressions (compare Rosen et al. [28]). Appropriate phis are also inserted.

The hoisted operations and newly created phis imply new leaders for their values in the blocks where they are placed, and these leaders must be propagated to dominated blocks. This could be done by re-running BuildSets, but that is unnecessary and costly. Instead, we assume a map which associates blocks with sets of expressions which have been added to the value leader sets during Insert. Whenever we create a new computation or phi, we possibly make a new

value, and we at least create a new leader for that value in the given block. We update that block's leader set and its new set. Since this information should be propagated to other blocks which the new temporaries reach, for each block we also add all the expressions in its dominator's new set into the block's own leader set and new set. This also requires us to make Insert a top-down traversal of the dominator tree, so that a block's dominator is processed before the block itself.

What we have said so far, however, is not completely safe in the sense that it in some cases it will lengthen a computational path by inserting an instruction without allowing one to be eliminated by the next phase. Since $v_3 + v_{100}$ is also anticipated in at block 6, being anti-inherited from $t_3 \leftarrow t_2 + 1$ in block 2, we insert $t_{14} \leftarrow t_3 + 1$ in block 4. This does not allow any eliminations, but only lengthens any trace through block 4. The value is still not available at the instruction that triggered this in block 1 because block 1 was visited first, at which time it was available on no predecessors and therefore not hoisted. To fix this, we repeat the process until we make a pass where nothing new is added to any new set. On the next pass, $v_7 : t_{15}$ is available in block 6, so we hoist $t_1 + 1$ to block 1. In practice, Insert converges quickly. We have seen only one case where it required 3 rounds. On most benchmarks, the maximum number of required rounds for any method was 2, and on average it took only a single round. Note that we do not need predicates for determining latest and earliest insertion points, since insertions naturally float to the right place. Insertions made too late in the program will themselves become redundant and eliminated in the next phase.

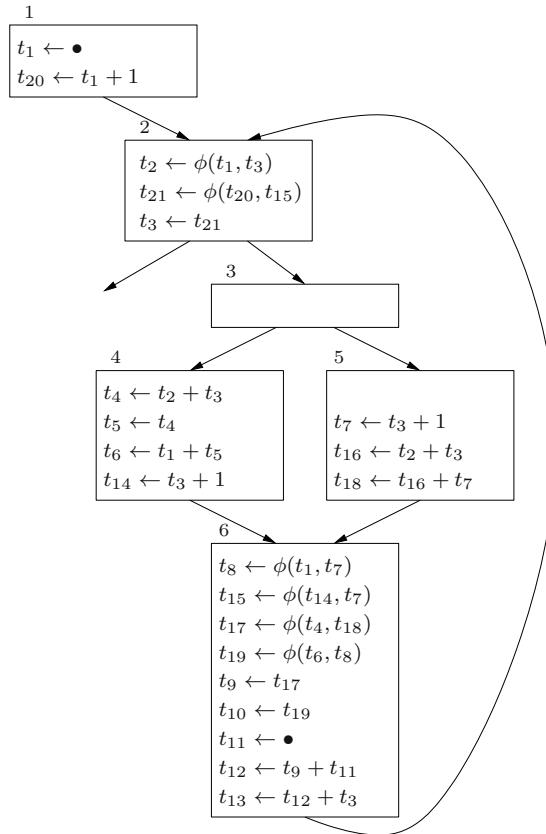
3.3 Eliminate

Eliminate is straightforward. For any instruction, find the leader of the target's value. If it differs from that target, then there is a constant or an earlier-defined temporary with the same value. The current instruction can be replaced by a move from the leader to the target. The order in which we process this does not matter. The optimized program is shown in Figure 5.

Corner cases. Certain extensions must be made to this algorithm for theoretical optimality (such as *code placement* in Knoop et al [21]). In practice, we have found them to yield no benefit, but they are described in a technical report [35].

4 Implementation and Results

We have proven the concept of this approach by implementing it in an open-source compiler; our implementation is not compared against other hybrid approaches which are difficult to implement, but is shown to subsume GVN. Our experiments use Jikes RVM [2,7,27], a virtual machine that executes Java class-files. We have implemented the algorithm described here as a compiler phase for the optimizing compiler and configured Jikes RVM version 2.3.0 to use the

**Fig. 5.** The optimized program

optimizing compiler only and a copy mark-sweep garbage collector. The optimizing compiler performs a series of code transformations on both SSA and non-SSA representations. It already has loop-invariant code motion (LICM) and global common subexpression elimination (GCSE) phases in SSA, which rely on GVN. Placing our phase before these two shows that GVN-PRE completely subsumes GCSE in practice. LICM is not completely subsumed because Jikes RVM's LICM performs unsafe speculative motion which GVN-PRE does not. GCSE is equivalent to GVN as it is presented in this paper.

Figure 6 shows static eliminations performed by each optimization level. The left column in each set represents the number of intermediate representation operations eliminated by GCSE, and the second those eliminated by GVN-PRE. In each case, GVN-PRE eliminates more, sometimes twice as much, although the optimization also inserts operations and in some cases may later eliminate an operation it has inserted. The third bar shows the number GVN-PRE eliminations plus the number of operations GCSE can eliminate after GVN-PRE

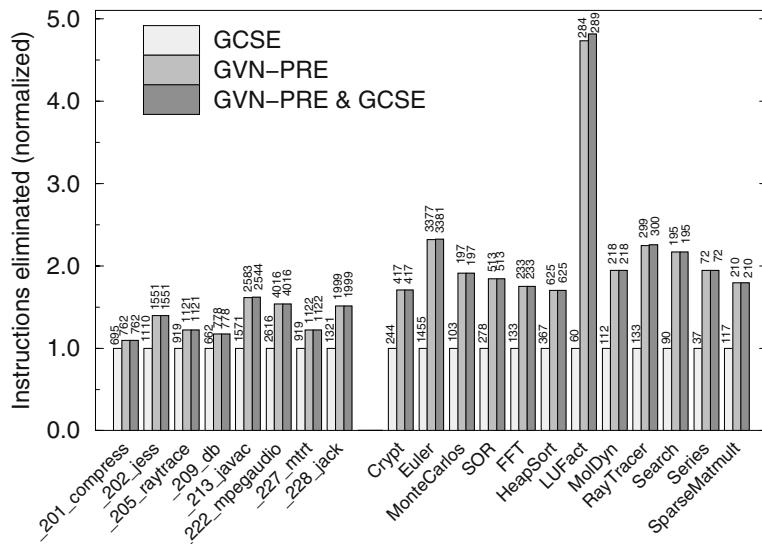


Fig. 6. Static results for the benchmarks

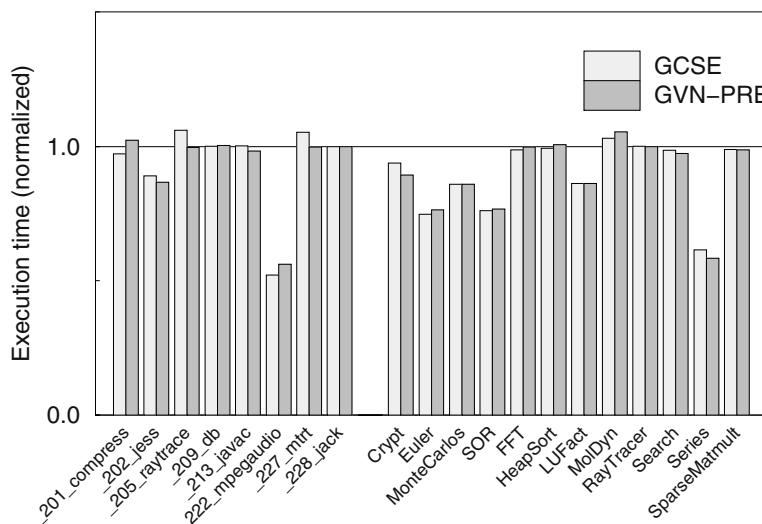


Fig. 7. Performance results for the benchmarks

has run. After GVN-PRE, GCSE finds zero in all but four cases, and never more than 6, compared to hundreds or thousands eliminated otherwise. Thus our optimization subsumes GCSE. All these are normalized to the first bar.

Our performance results are in Figure 7. These were obtained on a 1600MHz Intel Pentium 4 with 512 MB of RAM and 256 KB cache, running Red Hat Linux 3.2.2-5. We used three optimization levels: the pre-defined O2 with GCSE and GVN-PRE turned off, O2 with GCSE on but GVN-PRE off, and O2 with GCSE off but GVN-PRE on. In all cases, LICM and load elimination were turned off to isolate the impact of GCSE and GVN-PRE. We used twenty benchmarks in total, eight from the SPECjvm98 suite [10] and twelve from the sequential benchmarks of the Java Grande Forum [14]. For each benchmark at each optimization level we ran the program eleven times in the same invocation of the VM and timed the last nine runs using Java's `System.currentTimeMillis()` method. This way we did not incorporate compilation time in the measurements. We also subtracted garbage collection time. The graph shows the best running time of GCSE and GVN-PRE (out of the nine timed runs) normalized by the best running time of running with neither optimization. In only a few benchmarks is there significant performance gain, and sometimes these optimizations do no pay off. Studies have shown that in an environment like Jikes RVM, in order for an instruction elimination optimization to make a significant impact, it must address the removal of redundant loads and stores for objects and arrays [23], and we plan to extend our algorithm to incorporate those.

5 Conclusions and Future Work

We have presented an algorithm in which PRE and GVN are extended into a new approach that subsumes both, describing it formally as a dataflow problem and commenting on a practical implementation. This demonstrates that performing PRE as an extension of GVN can be done simply from a software engineering standpoint, and that it is feasible as a phase of an optimizing compiler. For future work, we plan to produce a version that will deliver real performance gains. Studies have shown that in an environment like Jikes RVM, in order for an instruction elimination optimization to make a significant impact, it must address the removal of redundant loads and stores for objects and arrays [23]. Accordingly, work to extend this algorithm for load and store instructions using Array SSA form [18,15] is underway.

Acknowledgments. The authors thank Di Ma for helpful conversation. We also thank the anonymous reviewers for their suggestions. This work is supported by the National Science Foundation under grants Nos. CCR-9711673, IIS-9988637, and CCR-0085792, by the Defense Advanced Research Program Agency, and by gifts from Sun Microsystems and IBM.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—principles, techniques, and tools*. Addison Wesley, 1986.
2. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.D. Choi, A. Cochchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.
3. Bowen Alpern, Mark Wegman, and Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.
4. Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the Symposium on Principles of Programming Languages* [26], pages 237–251.
5. Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 159–170, Orlando, FL, June 1994. SIGPLAN.
6. Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software—Practice and Experience*, 27(6):701–724, 1997.
7. Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, M Serrano, V Sreedhar, H Srinivasan, and J Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.
8. Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 273–286, Las Vegas, Nevada, June 1997.
9. Cliff Click. Global code motion, global value numbering. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 246–257, La Jolla, CA, June 1995. *SIGPLAN Notices* 30(6), June 1995.
10. Standard Performance Evaluation Council. SPEC JVM 98 benchmarks, 1998. <http://www.spec.org/jvm98/>.
11. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
12. Dhananjay M. Dhamdhere. E-path-pre—partial redundancy elimination made easy. *ACM SIGPLAN Notices*, 37(8):53–65, August 2002.
13. Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen’s “lazy code motion”. *ACM SIGPLAN Notices*, 28(5):29–38, May 1993.
14. EPCC. The java grande forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/index_1.html.
15. Stephen Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *Proceedings of the Static Analysis Symposium*, pages 155–174, Santa Barbara, California, July 2000.
16. Karthik Gargi. A sparse algorithm for predicated global value numbering. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 45–56, Berlin, Germany, June 2002.
17. Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, May 1999.

18. Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in Parallelizaiton. In Proceedings of the Symposium on Principles of Programming Languages [26], pages 107–120.
19. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 224–234, San Francisco, CA, July 1992. *SIGPLAN Notices* 27(7), July 1992.
20. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.
21. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Code motion and code placement: Just synonyms? In *Proceedings of the 7th European Symposium on Programming (ESOP)*, pages 154–169, Lisbon, Portugal, 1998. LNCS 1381.
22. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Expansion-based removal of semantic partial redundancies. In *Proceedings of the 8th International Conference on Compiler Construction (CC)*, pages 91–106, Amsterdam, The Netherlands, 1999. LNCS 1575.
23. Han Lee, Amer Diwan, and J. Eliot B. Moss. Understanding the behavior of compiler optimizations. Submitted.
24. Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.
25. Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
26. *Proceedings of the Symposium on Principles of Programming Languages*, San Diego, California, January 1998.
27. IBM Research. The Jikes Research Virtual Machine.
<http://www-124.ibm.com/developerworks/oss/jikesrvm/>.
28. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM Press, 1988.
29. Oliver Rüthing, Bernhard Steffen, and Jens Knoop. Detecting equalities of variables—combining efficiency with precision. In *Proceedings of the static analysis symposium (SAS)*, pages 232–247, 1999.
30. Loren Taylor Simpson. *Value-driven redundancy elimination*. PhD thesis, Rice University, 1996.
31. Bernhard Steffen. Optimal run-time optimization—proved by a new look at abstract interpretation. In *Proceedings of the Second International Joint conference on theory and practice of software development (TAPSOFT)*, pages 52–68, 1987. LNCS 249.
32. Bernhard Steffen, Jens Knoop, and Oliver Rüthing. The value flow graph—a program representation for optimal program transformations. In *Proceedings of the European Conference on Programming (ESOP)*, pages 389–405, 1990. LNCS 432.
33. Bernhard Steffen, Jens Knoop, and Oliver Rüthing. Efficient code motion and an adaptation to strength reduction. In *Proceedings of the sixth International Joint conference on theory and practice of software development (TAPSOFT)*, 1991. LNCS 494.
34. Thomas VanDrunen and Antony L Hosking. Anticipation-based partial redundancy elimination for static single assignment form. Submitted, 2003.
35. Thomas VanDrunen and Antony L Hosking. Corner cases in value-based partial redundancy elimination. Technical Report CSD-TR#03-032, Purdue University Department of Computer Sciences, 2003.

Increasing the Applicability of Scalar Replacement

Byoungro So¹ and Mary Hall²

¹ IBM T. J. Watson Research Center
1101 Kitchawan Road / Route 134, Yorktown Heights, NY 10598
bso@us.ibm.com

² University of Southern California / Information Sciences Institute
4676 Admiralty Way Suite 1001, Marina del Rey CA 90292
mhall@isi.edu

Abstract. This paper describes an algorithm for scalar replacement, which replaces repeated accesses to an array element with a scalar temporary. The element is accessed from a register rather than memory, thereby eliminating unnecessary memory accesses. A previous approach to this problem combines scalar replacement with a loop transformation called unroll-and-jam, whereby outer loops in a nest are unrolled, and the resulting duplicate inner loop bodies are fused together. The effect of unroll-and-jam is to bring opportunities for scalar replacement into inner loop bodies. In this paper, we describe an alternative approach that can exploit reuse opportunities *across multiple loops in a nest*, and *without requiring unroll-and-jam*. We also use this technique to *eliminate unnecessary writes* back to memory. The approach described in this paper is particularly well-suited to architectures with large register files and efficient mechanisms for register-to-register transfer. From our experimental results mapping 5 multimedia kernels to an FPGA platform, assuming 32 registers, we observe a 58 to 90 percent of reduction in memory accesses and speedup 2.34 to 7.31 over original programs.

1 Introduction

A standard strategy for reducing the high (and growing) cost of accessing external memory is to identify multiple memory accesses to the same memory location that *reuse* identical data, and replace unnecessary accesses with references to local storage close to the processor, *e.g.*, registers, caches, compiler-managed buffers. With respect to array variables in loop nest computations, a *data reuse* can be exploited when there are multiple references to an array element in the same or a subsequent iteration of a loop. The first such dynamic reference may either be a read or write; the reuse occurs on subsequent read references. A related optimization opportunity is to eliminate redundant writes to memory, where an array element is written, and on a subsequent iteration is overwritten; only the final value must be written back to memory. In this paper, we refer to this optimization as *register reuse*.

One specific code transformation for exploiting data reuse is *scalar replacement* [1], which replaces array references with scalar temporaries. As such, two references that access the same memory location will instead use the scalar temporaries. This idea can be extended to exploit register reuse by replacing all but the final write of a datum with writes to a register.

One previous approach to scalar replacement eliminates unnecessary read accesses based on true and input dependences in the innermost loop of a nest [2]. It is performed in conjunction with a transformation called *unroll-and-jam*, whereby outer loops in a nest are unrolled, and the resulting duplicate inner loop bodies are fused together. Unroll-and-jam can be used to decrease the distance in the loop nest's iteration space between two dependent references, and increase data reuse in the innermost loop. While this overall strategy has been shown to be very effective at speeding up important computational kernels, the reliance on unroll-and-jam has several limitations. Unroll-and-jam is not always legal, and because it significantly transforms the code, leading to larger loop bodies, it may conflict with other optimization goals.

In this paper, we describe a new approach for scalar replacement of array variables that extends this prior work in several important ways.

1. Increases the applicability by eliminating the necessity for unroll-and-jam;
2. Increases the candidates for reuse both by exploiting reuse across multiple loops in a nest (not just the innermost loop) and by removing redundant write memory accesses (in addition to redundant reads);
3. Provides a flexible strategy to trade off between exploiting reuse opportunities and reducing the register requirements of scalar replacement.

This new approach was motivated by the opportunities arising in compiling to FPGAs: (1) the amount of logic that can be configured for registers far exceeds the number of registers in a conventional system (on the order of several hundred to a few thousand); and, (2) data can be copied between registers in parallel. For these reasons, the compiler stores reused data in registers across loops in a nest, even though the reuse distance can be potentially large, and frequently copies between registers. Further, it is beneficial to avoid the code size increase associated with unroll-and-jam, since increased design complexity can lead to slower achievable clock rates and may exceed device capacity.

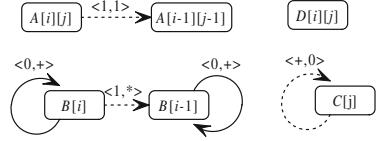
While FPGAs represent a unique computing platform, these extensions to scalar replacement will become increasingly important due to a number of current architectural trends, such as the larger register files of 64-bit machines (such as the Itanium's 128 registers), rotating register files [3], and software-managed on-chip buffers (e.g., Imagine [4]). The algorithm described in this paper has been fully implemented in the DEFACTO system, a design environment for mapping applications written in C to FPGA-based systems [5]. We present results on five multimedia kernels, demonstrating a 58 to 90 percent reduction in memory accesses and a speedup of 2.34 to 7.32 over the original programs.

The remainder of this paper is organized as follows. The next section presents background and motivation for this research and briefly introduces the prior solution by Carr and Kennedy [2] for purposes of comparison with our approach.

```

for (i = 1; i < 65; i++)
  for (j = 1; j < 33; j++)
    A[i][j]=A[i-1][j-1]+B[i]+
      B[i-1]+C[j]+D[i][j];
  
```

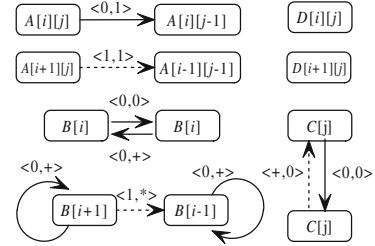
(a) Original code



(b) Reuse graph

```

for (i = 1; i < 65; i+=2)
  for (j = 1; j < 33; j++)
    A[i][j]=A[i-1][j-1]+B[i]+
      B[i-1]+C[j]+D[i][j];
    A[i+1][j]=A[i][j-1]+B[i+1]+
      B[i]+C[j]+D[i+1][j];
  
```

(c) After unroll-and-jam of loop i 

(d) Reuse graph after unroll-and-jam

Fig. 1. An example code

In Section 3, we describe the details of our algorithm, followed by a discussion of code generation in Section 4. Experimental results are presented in Section 5. Related research is discussed in Section 6, followed by a conclusion.

2 Background and Motivation

We use an example code in Fig. 1(a) throughout this paper to illustrate our approach. The write reference $A[i][j]$ and the read reference $A[i-1][j-1]$ in Fig. 1(a) represent a data reuse opportunity. The data written by $A[i][j]$ will be read by $A[i-1][j-1]$ in the next iteration of loop i and loop j . Similarly, the data fetched by $B[i]$ is read again by $B[i-1]$ in the next iteration of loop i . Both sets of references carry other data reuse opportunities in that the same data is accessed repeatedly in loop j . The read reference $C[j]$ can be reused repeatedly during the outer loop execution by keeping all portions of array C accessed by the inner loop in a series of registers, provided that there are enough registers available. Array reference $D[i][j]$ does not carry any data reuse opportunity.

A *dependence vector* $\mathbf{d} = \langle d_1, d_2, \dots, d_n \rangle$ refers to a vector difference of distance in an n -dimensional loop iteration space [6]. If the first non-zero entity of a dependence vector is positive, we call it *lexicographically positive*. Fig. 1(b) shows an example of a lexicographically positive dependence. We use D to refer to a set of dependence vectors.

For data reuse analysis, we only consider true, input, and output dependences. Anti-dependences are not considered as candidates of data reuse and are ignored. A dependence vector must also be lexicographically positive for the data reuse to be realizable. Under these requirements, a dependence vector entity which represents the difference in iteration counts between source and sink references of the dependence edge is one of the following:

- c : an integer constant distance.
- $+$: a set of distances that range from 1 to ∞ .
- $*$: a set of distances that range from $-\infty$ to ∞ , or an unknown distance.

A constant dependence vector entity c means that the distance between two dependent array references in the corresponding loop is c . A *loop-independent* data dependence whose vector entities are all zeros occurs when an array element is referenced multiple times in the same loop iteration. Since we are considering only lexicographically positive dependences, all dependence vectors must have as their first non-zero entity a positive integer, or ‘ $+$ ’.

Just like a dependence graph in dependence analysis, we represent a set of data reuse relations by a directed graph called a *data reuse graph*, which includes a set of reuse instances. A *reuse instance* consists of two dependent array references, one source and one sink, and a *reuse edge* which represents the representative dependence vector between the two array references. A reuse instance is referred to in the literature as *temporal reuse*. Temporal reuse from an array reference to itself is referred to as *self-temporal*, while temporal reuse among different array references is called *group-temporal*. A self-loop reuse edge \circlearrowleft in Fig. 1(b) represents self-temporal reuse across loop iterations. Array references with no incoming true or input dependence edges are called *reuse generators*. Array references with no outgoing output dependence edges are called *finalizers*. In Fig. 1(b), $A[i][j]$ in reuse edge $\{A[i][j] \rightarrow A[i-1][j-1]\}$ is both the reuse generator and the finalizer.

We briefly introduce a previous approach to scalar replacement by Carr and Kennedy [2] for the purposes of comparison with our approach. As stated earlier, their approach relies on unroll-and-jam to shorten reuse distance and thus reduce the number of registers required to exploit a reuse instance. (It is also used to increase instruction-level parallelism.) Carr and Kennedy’s approach restricts scalar replacement to array references with dependences carried by the innermost loop, thus of the form $\langle 0, 0, \dots, 0, d_n \rangle$. Further, each dependence must be *consistent*, which means that the dependence distance for the loop that carries the dependence is constant throughout the loop [6]. Unroll-and-jam is used to bring dependences carried by outer loops into the innermost loop. In Fig. 1(c), for example, the array references $A[i-1][j-1]$, $B[i-1]$, and $C[j]$ that induce the dependence vectors $\langle 1, 1 \rangle$, $\langle 1, * \rangle$, $\langle +, 0 \rangle$, identified by dotted arrows, are not of the form $\langle 0, 0, \dots, 0, d_n \rangle$. After unroll-and-jam, as in Fig. 1(c), some dependences are $\langle 0, 1 \rangle$, $\langle 0, 0 \rangle$, $\langle 0, + \rangle$ as illustrated in Fig. 1(d), increasing the innermost loop reuse. Similarly, the dependence $\langle +, 0 \rangle$ of array reference $C[j]$ in Fig. 1(b) is not carried by the innermost loop. By unrolling the outer loop, it creates a loop-independent dependence $\langle 0, 0 \rangle$.

The overall algorithm is to reduce unnecessary memory accesses and to match memory and floating point operations with the peak performance of the target architecture [2]. In this paper, M refers to the number of memory accesses remaining after scalar replacement, and R the number of registers required to exploit data reuse. As unroll factors increase, data reuse increases monotonically. However, R also increase and can exceed the number of available registers. Thus, the algorithm attempts to derive the best unroll factors computed as a function of M and the number of floating point operations under the constraint that R is less than the total number of available registers for a given architecture. Thus, the core of their analysis is to compute M and R , parameterized by the unroll factors u_1, \dots, u_{n-1} for loops 1 to $n-1$; the algorithm never unrolls the innermost loop because doing so does not affect the data access patterns.

However, their approach still cannot exploit some data reuse opportunities that the outer loop carries even after unroll-and-jam. Thus, array C and B in Fig. 1(c) repeatedly accesses the same array element across outer loops. Further, unroll-and-jam is not always safe, and may conflict with other compiler optimizations.

Compared to Carr and Kennedy’s approach, our approach does not require any further unrolling to expose data locality to the innermost loop. In addition, our approach does not have the limitation of applicable dependences of the form $\langle 0, 0, \dots, 0, d_n \rangle$. We show how we achieve these goals in the next section.

3 Extending Scalar Replacement

Since we do not perform unroll-and-jam, a key distinction in our approach is a strategy for analyzing reuse distance across multiple loops in a nest. In conjunction with the analysis, we perform register rotates and shifts, as discussed in the next section, to exploit this reuse. Another key difference is the use of reuse chains, rather than pairwise dependences, to identify and group together a collection of references that may refer to the same location. The effect of a reuse chain is that the number of registers needed to exploit the reuse can be reduced as compared to considering reuse among pairs of references. We consider output dependences, not just input and true dependences. As a consequence of this and the goal of exploiting reuse across multiple loops in a nest, the way in which we partition references is also different.

The steps of our algorithm are the following:

1. Perform the dependence analysis for a given code.
2. Build a reuse graph, where we partition all array references into reuse chains $C^\emptyset, C^c, C^{i\emptyset}$, and C^{ic} , as described below.
3. Compute R and M for each reuse chain. We compute M as part of register pressure control, as described in Section 3.3.
4. If needed to control register pressure, compute efficiency metric of each reuse chain, and tile some inner loops to tradeoff some data reuse opportunities with a smaller number of registers.
5. Replace array references with appropriate scalars.

6. Insert register shift/rotation statements to transfer a data item from the generator down to other references in a reuse chain.
7. Perform loop peeling and loop-invariant code motion to initialize or finalize registers for loop-invariant array references.

3.1 Definitions

A *reuse chain*, denoted as C , is a connected component of a reuse graph. A reuse generator provides the data that is used by all the references in the reuse chain, and the finalizer finalizes the value of the array element that may have been overwritten several times by previous write references.

Since we are exploiting data reuse across multiple loops in a nest, we need a concept of the distance between the references to the same array element across outer loop iterations. A *group temporal reuse distance* is defined by the minimal difference in iteration counts of the innermost loop between two dependent references. A reuse distance is different from a dependence distance in that it represents only the smallest dependence distance between the two dependent array references when there are multiple distances. For a given $\mathbf{d} = \langle c, \dots, c \rangle$ in an n -deep loop nest, we define the group temporal reuse distance $\epsilon(\mathbf{d}, n)$ as follows:

$$\epsilon(\mathbf{d}, n) = \sum_{l=1}^{n-1} \left\{ \left(\prod_{k=l+1}^n I_k \right) \times d_l \right\} + d_n, \quad (1)$$

where I_k is the iteration count of the k -th loop in a nest, and c is an integer constant. In Fig. 1(b), for instance, the reuse distance of a dependence vector $\langle 1, 1 \rangle$ of $\{A[i][j] \rightarrow A[i-1][j-1]\}$ is 33 ($= 1 \times 32 + 1$).

We partition reuse chains that exploit different kinds of data reuse across loop iterations into four categories; *i.e.*,

- C^\emptyset : reuse chains that carry no data reuse opportunities.
- C^c : reuse chains that carry only group-temporal reuse opportunities.
- $C^{i\emptyset}$: reuse chains that carry only self-temporal reuse opportunities.
- C^{ic} : reuse chains that carry both self-temporal and group-temporal reuse opportunities.

Reuse chain categories C^\emptyset and C^c can contain a loop independent data reuse among lexically identical array references in addition to the conditions above.

Our categorization is different from Carr's in two ways. First, since we are exploiting data reuse across outer loops as well as the innermost loop, we discriminate reuse chains that exploit data reuse only in invariant loops (category $C^{i\emptyset}$) from reuse chains that exploit data reuse in the invariant loops as well as with a constant dependence distance in some loops (category C^{ic}). Secondly, we categorize both read and write references, which allows us to eliminate redundant writes back to memory. Table 1 shows the characteristics of the dependences in each category and examples of reuse chains from Fig. 1(b). Note that we compute M and R throughout the entire execution of a loop nest.

Table 1. Comparison of reuse chain categories.

	C^\emptyset	C^c	$C^{i\emptyset}$	C^{ic}
D	\emptyset or $\{0, \dots, 0\}$	$\{\langle c, \dots, c \rangle\}$	$\{\langle \dots, +/*, \dots, 0, \dots \rangle\}$	$\{\langle \dots, +/*, \dots, c, \dots \rangle\}$
e.g.	$\{D[i][j]\}$	$\{A[i][j] \rightarrow A[i-1][j-1]\}$	$\{C[j] \circlearrowleft\}$	$\{\circlearrowleft B[i] \rightarrow B[i-1] \circlearrowleft\}$
G	2,048	2,048	32	64
A		1,953		63
M	2,048	2,143	32	65
R	0	34	32	2

In the next section, we use the following terms to describe the algorithm. \mathbf{A} refers to the group-temporal reuse associated with a particular incoming dependence or reuse chain; *i.e.*, how many memory accesses associated with one or more dependences can be eliminated. \mathbf{G} refers to the number of memory accesses incurred, associated with a reuse generator. In addition, we assume that there are enough registers to exploit all the data reuse opportunities exposed by data reuse analysis, and that the loop bounds are constant. We relax these assumptions by generalizing the algorithm in Section 3.3.

3.2 Computing R and M for Each Reuse Chain

We compute the number of registers and the number of memory accesses for each reuse chain, while Carr’s approach does so for each individual array reference. Because the data of a reuse generator may be used several times until the last reference in a reuse chain uses it, the dependence vector \mathbf{d} between the generator and the last reference in a reuse chain decides the required number of registers to fully exploit possible data reuse.

For a given \mathbf{d} and a set of loop iteration counts, we compute A and G for each data reuse chain as follows:

$$G = \prod_{l=1}^n \alpha(d_l, I_l), \text{ where } \alpha(d_l, x) = \begin{cases} x, & \text{if } d_l \text{ is a constant;} \\ 1, & \text{otherwise,} \end{cases} \quad (2)$$

$$A = \prod_{l=1}^n \beta(d_l, I_l), \text{ where } \beta(d_l, x) = \begin{cases} 1, & \text{if } d_l \text{ is not a constant;} \\ x - d_l, & \text{if } x \geq d_l; \\ 0, & \text{if } x < d_l. \end{cases} \quad (3)$$

The above equations are functions of I rather than unroll factors, since we exploit data reuse in the entire loop nest and do not unroll the loops. In Table 1, for example, $G=2,048$ and $A=1,953$ for reuse chain $\{A[i][j] \xrightarrow{\langle 1,1 \rangle} A[i-1][j-1]\}$.

In the rest of this subsection, we describe how to compute R and M for each type of reuse chain for a given dependence vector $\mathbf{d} = \langle d_1, d_2, \dots, d_n \rangle$ between the generator and the last array reference in the reuse chain.

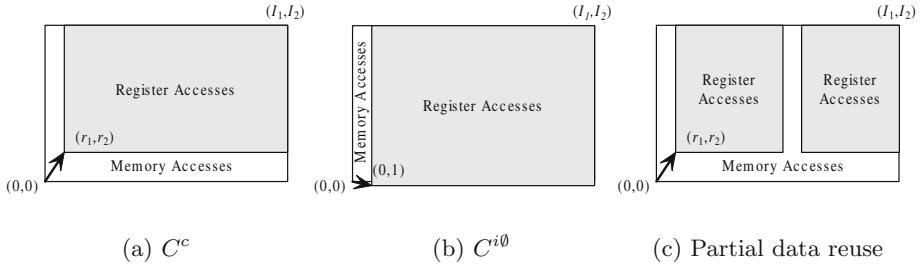


Fig. 2. Memory/register access behaviors of full and partial data reuse.

Reuse chains in C^\emptyset carry no reuse across iterations, but multiple references in the chain may carry loop-independent reuse within the loop body. In this case, we use one register to exploit loop-independent data reuse from the generator. Thus, M and R for reuse chains in C^\emptyset can be computed as follows:

$$M^\emptyset = G, \quad R^\emptyset = \begin{cases} 0, & \text{if } d = \emptyset \\ 1, & \text{if } d = \langle 0, \dots, 0 \rangle. \end{cases}$$

In Table 1, for example, $M^\emptyset=2,048$ and $R^\emptyset=0$ for reuse chain $\{D[i][j]\}$, since it does not exploit any data reuse.

Reuse chains in C^c captures only group-temporal reuse. Therefore, M and R required for reuse chains in C^c can be computed as follows:

$$M^c = 2G - A, \quad R^c = \epsilon(\mathbf{d}, n) + 1,$$

where the function $\epsilon(\mathbf{d}, n)$ is defined in Equation 1.

The reuse generator accesses memory in the entire iteration space, which leads to G memory accesses. Other references in the same reuse chain fetch data from memory only in the initial d_l iterations of each loop l . Figure 2(a) illustrates the general data access behavior for a reuse chain with reuse distance $\langle d_1, d_2 \rangle$. In the shaded region, which represents the group-temporal reuse amount A , the sink reference of a reuse instance gets necessary data from the source reference. However, the sink references access memory in the L-shaped white region. To exploit data reuse for reuse chains in C^c , a series of registers are necessary to keep the data until it is fully reused. The reuse generator fetches data from memory to the first register in a series, and each register A_i shifts the data to its neighboring register A_{i+1} in every innermost loop's iteration. Finally, after iterations of the innermost loop corresponding to the reuse distance $\epsilon(\mathbf{d}, n)$, the last register $A_{\epsilon(\mathbf{d}, n)}$ gets the necessary data. Meanwhile, the reuse generator keeps fetching a new datum from memory in every iteration; i.e., G memory accesses. The L-shaped white region can be computed by subtracting the shaded region from the whole region; i.e., $G - A$.

Therefore, the total number of memory accesses for a reuse chain in C^c is $G + G - A = 2G - A$, since the generators as well as other references belong to C^c . In Table 1, for example, $M^c=2,143$ and $R^c=34$ for reuse chain $\{A[i][j] \xrightarrow{\langle 1,1 \rangle} A[i-1][j-1]\}$.

Reuse chains in $C^{i\emptyset}$ have a dependence vector that consists of ‘+’, ‘*’, and 0. Let d_j be the first non-zero dependence distance, which should be ‘+’ in this reuse chain category; i.e., $\langle 0, \dots, 0, +, \dots \rangle$. Then, M and R required for each reuse chain in $C^{i\emptyset}$ can be computed as follows:

$$M^{i\emptyset} = G, \quad R^{i\emptyset} = \prod_{l=j}^n \alpha(d_l, I_l), \text{ where } d_j = '+'.$$

The data in a register can be reused without any additional registers during the entire execution of loops corresponding to ‘+’ or ‘*’. Figure 2(b) illustrates the general data access behavior for a reuse chain with dependence vector $\langle 0, + \rangle$. Array elements along the dimensions corresponding to the dependence distance 0 are kept in a series of registers for the outer loops corresponding to dependence distance ‘+’ or ‘*’ to exploit self-temporal reuse. In the dimensions corresponding to leading zero dependence distances (i.e., dimensions from 1 to $j-1$ if $j > 1$), the reuse chain does not exploit data reuse at all, so they do not affect the number of registers. In Table 1, for example, $M^{i\emptyset}=32$ and $R^{i\emptyset}=32$ for reuse chain $\{C[j] \circlearrowleft \langle +, 0 \rangle\}$.

Reuse chains in C^{ic} contain self-loop edges and data reuse edges between different array references. As such, a dependence vector of reuse chains in C^{ic} consists of ‘+’, ‘*’, and constants. In this category, at least one constant should be non-zero.

Let d_j be the first non-constant dependence distance; i.e., $\langle c, \dots, c, +/*, \dots \rangle$. Then, the number of memory accesses remaining after scalar replacement for reuse chains in C^{ic} can be computed as follows:

$$M^{ic} = 2G - A, \quad R^{ic} = \{\epsilon(d, j-1) + 1\} \times \prod_{l=j}^n \alpha(d_l, I_l).$$

In Table 1, for example, reuse chain $\{\circlearrowleft B[i] \xrightarrow{\langle 1,* \rangle} B[i-1] \circlearrowleft\}$ can exploit self-temporal data reuse in the inner loop (corresponding to ‘*’) and group-temporal data reuse in the outer loop (corresponding to ‘1’). Since two references are invariant in the inner loop, we can determine the reuse distance in terms of the outer loop, which is 1. Thus, $R^{ic}=1+1=2$ for this chain.

Consider dependence vector $\langle +, 1 \rangle$. A data item is reused in the next iteration of the inner loop. In addition, the data is reused during the entire iterations of the outer loop. Thus, the same number of registers as the inner loop iteration count are necessary, no matter how small c is. Therefore, the first non-constant dependence distance in dimension j means that exploiting self-temporal reuse in j through n dimensions requires the following number of registers just like $R^{i\emptyset}$; i.e., $\prod_{l=j}^n \alpha(d_l, I_l)$.

```

for (i = 1; i < 65; i++) {
    B0 = B[i];
    if (i==1) B1 = B[i-1];
    for (j = 1; j < 33; j++) {
        if (i==1)
            { A33 = A[i-1][j-1]; C0 = C[j]; }
        else if (j==1) A33 = A[i-1][j-1];
        A0 = A33 + B0 + B1 + C0 + D[i][j];
        A[i][j] = A0;
        shift_registers(A0, ..., A33);
        rotate_registers(C0, ..., C31);
    }
    B1 = B0;
}

```



```

for (jt = 1; jt < 33; jt+=16)
    for (i = 1; i < 65; i++) {
        B0 = B[i];
        if (i==1) B1 = B[i-1];
        for (j = jt; j < jt+16; j++) {
            if (i==1)
                { A17 = A[i-1][j-1]; C0 = C[j]; }
            else if (j==jt) A17 = A[i-1][j-1];
            A0 = A17 + B0 + B1 + C0 + D[i][j];
            A[i][j] = A0;
            shift_registers(A0, ..., A17);
            rotate_registers(C0, ..., C15);
        }
        B1 = B0;
    }
}

```

(a) Full data reuse

(b) Partial data reuse

Fig. 3. Final output of our approach to scalar replacement.

Further, the constant dependence vector dimensions in the outer loops before dimension j require $\epsilon(\mathbf{d}, j-1)+1$ sets $R^{i\emptyset}$ registers in a similar way as computing R^c . For example, consider a dependence vector $\langle 1, *, 2 \rangle$ between two references $A[i][k]$ and $A[i-1][k-2]$ within a 3-deep loop nest. To exploit self-temporal reuse in the second dimension, I_k registers are required to keep the array elements accessed in the innermost loop. Further, since the dependence distance in the first dimension is 1, the total number of registers required for this reuse chain ends up multiplying 2 by I_k . Intuitively, this corresponds to I_k registers to exploit self-temporal reuse in the second dimension, and an additional I_k registers to exploit group-temporal reuse in the first dimension. Thus, $G=64$, $A=63$, $M^{ic}=65$, and $R^{ic}=2$ for reuse chain $\{\circlearrowleft B[i] \xrightarrow{\langle 1,* \rangle} B[i-1]\circlearrowright\}$.

Fig. 3(a) shows the result of our scalar replacement applied to the code in Fig. 1(a). The code in Fig. 3(a) does not access any array element more than once, fully exploiting data reuse opportunities at the cost of more registers used than Carr's approach. In the next subsection, we present how we control the register pressure, not losing many opportunities of data reuse.

3.3 Generalizing the Algorithm

In architectures where there are not enough registers to exploit all possible data reuse opportunities, we have to reduce the required number of registers to avoid register spills. Further, a non-constant loop bound hinders the reuse analysis from computing the reuse distance $\epsilon(d, n)$ in Equation 1. Fortunately, we can use the same technique to deal with these issues.

Partial data reuse trades off data reuse opportunities for lower register pressure. We exploit partial data reuse using a code transformation called *tiling*, which divides computation and data into blocks, and makes the data reuse closer in time. Partial data reuse exploits data reuse only within a tile and introduces

memory accesses across tiles. Fig. 2(c) shows the difference in memory access behaviors between full and partial data reuse. The loop to be tiled and the tiling factors are decided based on the dependence information. Tiling each loop i by T_i will reduce the reuse distance by

$$\sum_{j=1}^{n-1} \left\{ \left(\prod_{k=j+1}^n I_k \right) \times d_j \right\} - \sum_{j=1}^{n-1} \left\{ \left(\prod_{k=j+1}^n T_k \right) \times d_j \right\}.$$

From this tiled reuse distance, we can compute the number of registers and the number of memory accesses required for the tiled code. Fig. 3(b) shows the code after tiling the inner loop j , which requires 36 registers for partial data reuse and incurs 4,416 memory accesses.

Using tiling, we can also derive inner loop nests with constant loop bounds, when some bounds are not constant. If the iteration count is not divisible by the tiling factor, we can use *Index set splitting*, which divides the index set of a loop into multiple portions, replicating the body of the loop as appropriate [6]. We use index set splitting to isolate the residual ($I_k \bmod T_k$) iterations from the main tiled loop l_k . As a result, each tile of iterations contains a fixed constant T_k iteration count.

Tiling is not always a legal transformation¹. An alternative way to control register pressure is *selective data reuse* which selectively exploits data reuse for some reuse chains depending on their register requirements and reuse benefits. Thus, we could alternatively compute the register efficiency for each reuse chain c as follows:

$$\text{Efficiency}_c = \overline{M}_c / R_c,$$

where \overline{M}_c refers to the number of memory accesses eliminated by scalar replacement and R_c the number of registers necessary for reuse chain c . A hybrid approach could apply selective data reuse to filter out extremely low efficiency reuse chains, and apply partial data reuse for the rest.

4 Code Transformation

Once the number of registers is computed, a series of scalar variables, S_0, S_1, \dots, S_{R-1} , are introduced, where R is the number of registers to be used to replace array references in a reuse chain. An array reference is replaced with an appropriate scalar variable depending on the reuse distance between its generator and the array reference within the same reuse chain. Each reuse generator that is a read reference needs a register initialization “ $S_i = A[]$ ” before the register is used, and the finalizer reference in a reuse chain requires a memory store “ $A[] = S_i$ ”. Intuitively, the insertion position of these initialization/finalization statements is the innermost loop in which the array reference is not invariant, thereby accomplishing loop-invariant code motion. A discussion of control flow is beyond the scope of this paper, but modifications to support control flow are described in [7].

¹ Its legality test is equivalent to unroll-and-jam.

4.1 Register Shift / Rotation

A reference receives data from its generator after the number of iterations corresponding to its reuse distance. Let r represent the reuse distance between the generator and the last array reference in a reuse chain. Then, to achieve reuse, a `shift_registers` operation, equivalent to a series of register copy statements $\{(S_{i+1} = S_i) \mid r - 1 \geq i \geq 0\}$, is inserted at the appropriate position as in Fig. 3. The insertion position of register shifts for each reuse chain can be decided by the same method as is used to determine the initialization/finalization point of registers.

In the case of reuse chain categories $C^{i\emptyset}$ and C^{ic} , data reuse occurs repeatedly in loops where the array reference is invariant. The `rotate_registers` operation shifts the data in a series of registers and rotates the last one into the first position.

4.2 Loop Peeling and Code Motion

Loop peeling removes the first (or last) several iterations of the loop body and makes them as separate code before (or after) loop execution. The scalar replacement optimization uses loop peeling in combination with loop-invariant code motion for register initialization and finalization purposes.

We see in Fig. 3(a) that values for register A_{33} are initialized on the first iteration of loop j and on the first iteration of loop i . For clarity it is not shown here, but the code generated by our compiler actually peels the first iteration of loop j and loop i instead of including these conditional loads so that the main body of both loops have the same number of memory accesses. Further, we can avoid the overhead of condition checks in every loop iteration. In addition, array reference $B[i]$ is invariant with respect to loop j , so the initialization of register B_0 and B_1 are moved outside the inner loop j . Within the main unrolled loop body, only a read reference to array D and a write reference to array A remain.

Our approach performs loop peeling on multiple loops in a nest. To initialize the scalar variables at the beginning of the loop iteration, each loop l_i needs to be peeled by the maximum dependence distance among all true and input dependence vectors. To finalize the array elements at the end of the loop iteration, each loop l_i needs to be peeled by the maximum dependence distance among all output dependence vectors. However, since we peel each loop by the maximal dependence distance among all dependence vectors, some array references whose dependence distance is less than the peel factor do not need to be initialized because they are lexically identical array references in some peeled iterations.

5 Experiment

This section presents experimental results that characterize the impact of the scalar replacement algorithm for a set of application kernels written in C: a digital FIR filter (FIR), matrix multiply (MM), pattern matching (PAT), Jacobi 4-point stencil (JAC), and Sobel edge detection (SOBEL). MM is implemented by a 3-deep loop nest, and the others are 2-deep loop nests.

Table 2. Problem size (iteration count).

Problem size	FIR			MM			PAT			JAC			SOBEL		
	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Outer loop	64	64	640	64	64	640	64	64	640	32	32	320	64	64	640
Middle loop				5	6	6									
Inner loop	30	32	60	5	6	16	31	32	62	15	16	30	14	16	28

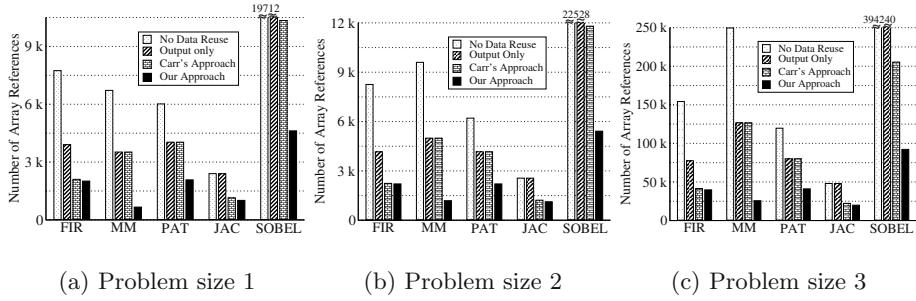
5.1 Methodology

The algorithm described in this paper has been fully implemented in the DEFACTO system, a design environment for mapping C applications to application-specific FPGA hardware implementations [5]. As compared to a conventional architecture, FPGAs have no instruction or data cache, and the microarchitecture is configured specifically for the application. DEFACTO is built on top of the Stanford SUIF compiler, in which we have implemented the scalar replacement transformation. After the compiler applies its transformations, DEFACTO automatically generates a VHDL hardware description of the algorithm, which is then synthesized to hardware by commercially available FPGA synthesis tools; the measurements provided in this section were obtained from Mentor Monet [8], a behavioral synthesis tool. While the output of Monet is not identical to the results obtained from executing the application in hardware, the number of cycles is the same and can be compared directly; other metrics have been shown to be consistent between Monet and the final implementation [9]. The target FPGA device is a Xilinx Virtex XCV 1000 [10].

In this experiment, we compare four data reuse schemes: (1) no data reuse, (2) redundant write elimination only, (3) an approximation to Carr's approach, and (4) our approach. The approximation to Carr's approach, selects the maximal unroll factor for the outer loop of the 2-deep loop nests such that the number of registers is not exceeded. Inner loops are not unrolled in Carr's algorithm. In the case of MM, we also unrolled the outermost loop since the results would be equivalent for either of the two outer loops. While Carr's algorithm might decide not to unroll this much, these results can be thought of as an upper bound on the benefits from Carr's scalar replacement algorithm, since the measurements for performance improvements on our FPGA system are not affected by code size.

To characterize the benefits and cost of each data reuse scheme, we measured four metrics: (1) the number of memory accesses remaining after each data reuse scheme, and (2) the number of registers used to exploit data reuse, and (3) the speedup over original programs, and (4) the FPGA space usage.

The first and second metrics heavily depend on the iteration counts of the loops. For each program, thus, we compare three different problem sizes in terms of iteration count of each loop in a nest as shown in Table 2. We assume there are 32 registers available in the target architecture. The first problem size for each program requires less than or equal to 32 registers to fully exploit data

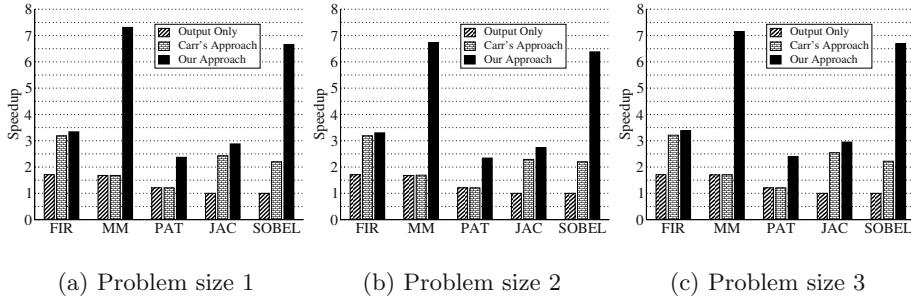
**Fig. 4.** Number of Memory Accesses.

reuse using our approach. The second size is slightly bigger than the first size, and thus it requires slightly more registers. The third size requires much more registers than 32 to exploit full data reuse. Thus, we perform partial data reuse in the cases of the second and third problem sizes of each program. For Carr’s approach, we maximize the unroll factor of the outer loop (and jam the copies of the inner loop) such that it uses less than or equal to 32 registers. Our approach is fully automated, but we derived results for output elimination only and Carr’s approach with some manual direction of our compiler.

5.2 Experimental Results

In Fig. 4, we measured how effectively we eliminated the quantity of redundant memory accesses, with each bar capturing results for one of the four previously described schemes. In Fig. 4(a) and (b), the second bar eliminated about 50% of memory accesses for FIR and MM, 33% for PAT. However, there was no opportunity for redundant write elimination at all for JAC and SOBEL. In Fig. 4(c), the second bar eliminated slightly more memory accesses (up to 1%), since the problem size is bigger. For three different problem sizes, Carr’s approach eliminated 72% of memory accesses for FIR, 52% for JAC, and 47% for SOBEL. However, it eliminated the same amount of redundant memory accesses as the second bar did for MM and PAT. On the other hand, our approach eliminated slightly more redundant memory accesses (up to 5% more) than Carr’s approach for FIR and JAC. However, it eliminated 90% of memory accesses for MM, 65% for PAT, and 77% for SOBEL in Fig. 4(a), and up to 3% less in Fig. 4(b), and up to 1% more in Fig. 4(c). If the problem size is bigger, the benefits of our approach to scalar replacement would be greater.

Table 3 shows the number of registers required to exploit data reuse for three data reuse schemes. Carr’s approach uses 32 registers for FIR, MM, and PAT, but 30 registers for JAC and SOBEL because increasing the unroll factor requires more than 32 registers. Only one register is required to eliminate redundant write accesses for FIR, MM, and PAT. There are no redundant array writes for JAC or SOBEL. Our approach uses a comparable number of registers for problem sizes 1 and 3, but considerably less registers for problem size 2 because tiling reduces

**Fig. 5.** Speedups of scalar replacement.

the iteration count of the inner loop. However, as shown in Figure 4(b), partial data reuse eliminated more redundant array accesses than in the (c) case, which uses many more registers.

Fig. 5 presents the speedup results of overall performance on a single FPGA, again corresponding to the four schemes above. The approximation to Carr’s approach observes speedups from 1.21 to 3.21, and output elimination only observes speedups from 1.21 to 1.71. On the other hand, our approach observes speedups from 2.34 to 7.31 for five programs. If we use full data reuse, which is possible on the FPGA platform for the smaller problem sizes, the speedups of our approach would be greater.

In summary, from our experiments, we observe several sources of benefits. While most benefits come from exploiting reuse of input and true dependences, redundant write elimination is valuable and reduces the number of memory accesses up to 51%. Exploiting data reuse across multiple loops does not necessarily require a large number of registers relative to the footprint of the accessed data, and results in the best performance. Partial data reuse within a limited number of registers eliminated up to 40% more redundant array accesses than our approximation to Carr’s approach. As a final benefit of our approach over using unroll-and-jam, the FPGA space required to implement the unrolled computation may be prohibitive. Larger designs have more routing complexity, and may lead to lower achieved clock rates. For the example kernels using the maximal unroll factors, the designs resulting from our algorithm were up to 80% smaller.

Table 3. Number of registers.

Problem size	FIR			MM			PAT			JAC			SOBEL		
	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Carr’s appr.	32	32	32	32	32	32	32	32	32	30	30	30	30	30	30
Output only	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
Our appr.	31	17	31	30	19	29	32	17	32	31	17	31	32	20	32

6 Related Work

The most closely related work [2] is explained in detail in this paper. While there have been numerous other works on data locality optimizations, we cite only those most closely related to our transformations here. Fink et al. [11] introduces a scalar replacement algorithm that can handle both array and pointer objects. In their model, they treat both pointer and array references as accesses to elements of hypothetical heap arrays and they can avoid an expensive point-to analysis. McKinley et al. [12] showed a compound data locality optimization algorithm to improve cache performance that place the loop that carries the most data reuse at the innermost position for each statement in a loop nest. Kolson et al. [13] uses an *internal data forwarding* technique to eliminate redundant memory traffic in a pipelining scheduler. Kodukula et al. [14] introduces *data shackling*, a data-centric approach to locality enhancement for L2 cache. It fixes a block of data and then determines which computations should be executed on it. Deitz et al. [15] introduces *array subexpression elimination* that eliminates redundant computation and accompanying memory accesses.

7 Conclusion

In this paper, we have described an algorithm for scalar replacement, used to eliminate redundant memory accesses by replacing array references with scalar temporaries. Our approach extends scalar replacement (1) to increase the applicability by eliminating the necessity for unroll-and-jam, and (2) to increase the candidates for reuse both by exploiting reuse across multiple loops in a nest (not just the innermost loop) and by removing redundant write memory accesses (in addition to redundant reads), and (3) to provide a flexible strategy to trade off between exploiting reuse opportunities and reducing the register requirements of scalar replacement. Using less than or equal to 32 registers for this technique, we observe a 58 to 90 percent of reduction in memory accesses and speedup of 2.34 to 7.31 over the original programs.

References

1. Callahan, D., Carr, S., Kennedy, K.: Improving register allocation for subscripted variables. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, New York, ACM press (1990)
2. Carr, S., Kennedy, K.: Improving the ratio of memory operations to floating-point operations in loops. ACM Transactions on Programming Languages and Systems **16** (1994) 1768–1810
3. Ng, J., Kulkarni, D., Li, W., Cox, R., Bobholz, S.: Inter-procedural loop fusion, array contraction and rotation. In: The 12th International Conference on Parallel Architectures and Compilation Techniques, New Orleans, LA (2003) 114–124
4. Rixner, S., Dally, W.J., Kapasi, U.J., Khailany, B., Lopez-Lagunas, A., Mattson, P.R., Owens, J.D.: A bandwidth-efficient architecture for media processing. In: International Symposium on Microarchitecture. (1998) 3–13

5. So, B., Hall, M.W., Diniz, P.C.: A compiler approach to fast hardware design space exploration in fpga-based systems. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02), New York, ACM press (2002) 165–176
6. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, San Francisco (2002)
7. So, B.: An Efficient Design Space Exploration for Balance between Computation and Memory. Ph.d. dissertation, University of Southern California, Los Angeles, CA (2003)
8. Mentor Graphics Inc.: MonetTM. R44 edn. (1999)
9. So, B., Diniz, P., Hall, M.: Using estimates from behavioral synthesis tools in compiler-directed design space exploration. In: The 40th Design Automation Conference, Anaheim, CA (2003)
10. XILINX: Virtex-II 1.5V FPGA Complete Data Sheet. DS031(v1.7), 2100 Logic Drive, San Jose, Calif. (2001)
11. Fink, S.J., Knobe, K., Sarkar, V.: Unified analysis of array and object references in strongly typed languages. In: Proceedings of the 2000 Static Analysis Symposium. (2000) 155–174
12. Mckinley, K.S., Carr, S., wen Tseng, C.: Improving data locality with loop transformations. ACM Transactions on Programming Languages and Systems **18** (1996) 424–453
13. Kolson, D., Nicolau, A., Dutt, N.: Elimination of redundant memory traffic in high-level synthesis. IEEE Trans. on Computer-aided Design **15** (1996) 1354–1363
14. Kodukula, I., Pingali, K., Cox, R., Maydan, D.E.: An experimental evaluation of tiling and shackling for memory hierarchy management. In: Proceedings of the ACM International Conference on Supercomputing. (1999) 482–491
15. Deitz, S.J., Chamberlain, B.L., Snyder, L.: Eliminating redundancies in sum-of-product array computations. In: Proceedings of the ACM International Conference on Supercomputing. (2001) 65–77

Reducing the Cost of Object Boxing

Tim Owen and Des Watson

Department of Informatics, University of Sussex, Brighton, UK

Abstract. Language implementations that use a uniform pointer representation for generic datatypes typically apply boxing operations to convert non-uniform objects into the required form. The cost of this boxing process impacts upon the performance of programs that make heavy use of genericity with non-uniform data such as integers and other primitive value types. We show that the overhead of boxing objects into heap storage can be significantly reduced by taking a lazy approach: allowing pointers to stack-allocated objects that are only copied to the heap when necessary. Delaying the boxing of objects avoids unnecessary heap allocation, and results in speedups of around 25% for a range of test programs.

1 Introduction

The implementation of languages supporting genericity typically use a uniform representation for generic data: a pointer to the actual object. This approach works well for objects that are of *reference type*, since they are already represented as a pointer to a heap allocated object. Objects of *value type*, however, are handled directly, can be allocated on the stack and passed by value. For example, integers and real numbers are value types in most languages. Since no pointer indirection is used, value type objects are not in the uniform representation required for use by generic code.

A common solution to this mismatch is for the compiler to insert a *boxing* coercion operation into the generated code that copies a value type object from the stack onto the heap. A pointer to this copy is then used by generic code. For example, when using a generic List to store integers, each integer will be boxed when passed to the code that implements the List data structure. A similar approach has also been applied in the C# language[1] to allow code expecting reference type data to be used for integers and other value types, by automatically boxing and unboxing objects where necessary.

Whilst the use of boxing solves the problem of achieving a uniform representation, it incurs an overhead in terms of extra memory used to store the boxed copy, and a performance reduction due to the task of copying objects from stack to heap. In addition, the extra use of heap storage increases the load on the garbage collector, which may cause further slowdowns. Therefore, it is desirable to either reduce the cost of boxing or even eliminate it altogether where possible.

In this paper, we introduce an optimisation called *lazy boxing* in which value type objects (including primitive type objects such as integers) are not immediately copied to the heap when passed to a generic context. Instead, generic

code uses a pointer to the object stored on the stack. A boxing operation is only performed later if the stack object is about to be deallocated. We discuss the implementation details of this technique, and show experimental results that demonstrate its effectiveness.

2 Related Work

Languages with genericity have different approaches to the support of value types in a generic context. Earlier proposals for extending Java with genericity, such as GJ[2] and NextGen[3], do not allow Java's value types (e.g. primitive types such as `int`) to be used as type arguments — only reference types are permitted. For example, GJ allows `List<Integer>` but not `List<int>` which simplifies their compiler implementation, but effectively requires the programmer to manually insert boxing and unboxing code. In our work, we allow generic types to work with all types of data, including value types such as integers.

While Java has been extended with support for genericity, first through research projects such as Pizza[4] and GJ and now officially in the 1.5 release, the language still does not support user-defined value types. The C# language allows programmers to define `struct` objects that are handled by value, and specifies that automatic boxing takes place when these objects are handled by reference.

A proposal for extending the Microsoft Common Language Runtime (CLR) with genericity[5] enables integrated support for value classes and genericity, by using a virtual machine infrastructure that builds type-specific information and generates specialised method code at class load-time. This avoids the need for boxing, but at the expense of increased code size and a more complex run-time support system. While their implementation follows one set of design choices and trade-offs, our work examines a different point in the design space. We take a fully-compiled approach, supporting value types and genericity by using boxing, but without need for sophisticated run-time code generation or specialisation.

Templates in C++ provide genericity[6], and may be used with value or reference types. Like the generic CLR, this approach completely avoids the need for object boxing, but with the disadvantage that it involves compile-time (or possibly load-time) expansion of generic definitions, which can result in undesirable code duplication. Besides C++, Eiffel also provides genericity and value types, although implementation details are hard to come by. The design of the SmallEiffel compiler[7] suggests that separate code is produced for different parameterisations of a generic class, with a common instantiation for all pointer-based types. Again, this avoids the need for boxing at the expense of duplicated code.

The Pizza project (an early extension of Java with generics) permits use of value types with generic types, and demonstrates a choice of implementation: either the heterogeneous approach of generating separate code for each use of a generic type, or the homogenous technique that shares a single code but introduces boxing and unboxing operations as required.

In the functional language community, the problem of supporting different sized data in a generic context is known as unboxed polymorphism. The aim is to avoid using the uniform representation (i.e. a pointer to a heap-allocated object) as much as possible. Coercion-based solutions[8,9] apply boxing and unboxing operations at run-time, whenever value type objects are passed to and from a generic context. This is the conventional eager boxing strategy. One limitation of the straightforward coercion approach is that the boxing operation for recursive data structures such as linked lists and trees can involve large amounts of expensive and redundant boxing. Passing type information at run-time can alleviate this problem[10], and has been applied to the FLINT[11] and TIL[12] compilers for ML.

Compile-time techniques have been developed that can be used to eliminate redundant boxing by analysis of the program code. Most notably, escape analysis[13,14] determines the extent of references to an object. This information can then be used to safely stack-allocate some objects, when the analysis is able to determine that no references to the object will outlive the object itself. Our work is complementary to this, in that we are interested in handling the boxing of those objects for which escape analysis *cannot* prove is redundant. Since escape analysis is conservative, it will leave a boxing operation in place when references to an object may escape. We are interested in optimising these remaining cases, by delaying the boxing until it is actually necessary. The descriptions of systems that we have reviewed suggest that when analysis cannot remove the need to box an object then an eager boxing operation is then used.

In summary, existing approaches to implementing the combination of generics and value types typically take one of two paths:

- Generate separate code for each different use of a generic type, thereby avoiding boxing completely — C++, Eiffel, Generic C#/CLR and Pizza (heterogeneous implementation) take this approach.
- Share a common code implementation, but insert boxing coercions as required to convert between the two representations — this is done by the ML compilers cited above and in Pizza (homogenous implementation).

However, the nature of the boxing in this second method of implementation is that it is applied eagerly. Our work refines this approach by delaying the boxing where possible.

3 The Pluto Language and Implementation

In order to examine the costs associated with supporting genericity and value type objects, we devised a prototype object-oriented language called Pluto and implemented a compiler for it[15]. This language supports the definition of value types and provides genericity in the form of parameterised types. The compiler implements the lazy boxing optimisation (described in Sect. 4) which can be switched on or off. This enables us to examine the effectiveness of the technique, and to measure its effect empirically.

The Pluto language is essentially similar to a generic version of Java, such as GJ[2] or NextGen[3], but with the addition of support for user-defined value types. The example code in Fig. 1 shows a value class called `point` and a generic `Cell` class. All of the code in the figure is programmer-level class definitions, rather than pre-defined. Therefore, a value class like `point` is similar to a struct in C#.

3.1 Value Types

A value class is akin to Eiffel’s expanded classes[16] or a struct in C#[1], so its instances are handled directly and copied by value. In Pluto, value classes are part of the class hierarchy and can be used in generic situations just like ordinary reference-based classes. To provide uniformity, there are no primitive types in the language syntax — simple types such as integers and booleans are represented at the language level as value classes, although for efficiency reasons their implementation is optimised by the compiler.

In keeping with conventional value semantics, all value classes are *immutable* so instances of a value class cannot change their internal state. This property is critical to ensure that the compiler can make boxed copies of value class objects without concerns over aliasing and visible state changes. While the objects of a value class are immutable, this does not imply that we disallow updating of variables. Indeed, the Pluto language is still imperative in the sense that a variable of value type can be updated to hold a new value object — but we do not allow the internal state of the objects themselves to be modified. For example, the `point` class in Fig. 1 is a value type, so its internal fields `x` and `y` cannot be updated. But the local variables `p` and `q` in the `Main.run` method can be updated to hold a different `point` object.

Apart from the restriction of immutability, value classes can be used in much the same way as reference classes: they can define methods, have subclasses and implement interfaces. Most specifically for the work discussed here, value classes can be used as types for generic type parameters, e.g. `List[point]` or `Cell[point]`.

3.2 Implementation Overview

Our compiler generates C code, using the traditional representation of objects as a C `struct` containing the instance fields. Instance methods are translated into functions with an extra parameter for the `this` object. Instances of reference type classes are heap-allocated and handled via a pointer, whereas value class instances are handled directly: C structs and the C primitive types such as `int` that are used directly for Pluto’s integer types, are passed around and copied by value.

For example, the code generated for the `Main.run` method in Fig. 1 will declare the two `point` objects `p` and `q` as local variables with a struct type, so they will be stored in the function’s stack frame. Since the `point` class is declared final, then the C structure for objects of this class contains just two

```

interface Comparable [E] {
    abstract method ==(other:E) : boolean;
}

final value class point(x:int, y:int) implements Comparable[point] {
    field x:int = x;
    field y:int = y;

    method ==(other:point) : boolean {
        return (this.x == other.x) && (this.y == other.y);
    }
}

class Cell [T <: Comparable[T]] (initialData:T) {
    field element:T = initialData;

    method get() : T {
        return this.element;
    }

    method set(newData:T) {
        var this.element = newData;
    }

    method elementEquals(otherData:T) : boolean {
        return (this.element == otherData);
    }
}

class Main() {
    static method run() : int {
        let p:point = new point(3,4);
        let c:Cell[point] = new Cell[point](p);
        let q:point = new point(3,4);
        if (c.elementEquals(q)) {
            String.println("c is holding a point equal to q");
        }
        return 0;
    }
}

```

Fig. 1. Example Pluto code

integer members. The `Cell` class is a reference class, so object `c` in the example code will be allocated on the heap and only a pointer to it will be stored in the stack frame.

We are assuming a traditional C implementation of memory management, with local variables stored in stack frames and heap allocation through the `malloc` library function. Our generated code uses an off-the-shelf conservative garbage collector library[17] to manage the heap. Since we are generating C code, the issue of managing storage for value type objects is largely down to the C compiler used, which may choose to store small value objects in registers rather than stack frames. Naturally this reliance on C and a conservative garbage collector means that we give up some control on the handling of object storage. In future work we could consider generating assembly code directly, or using a lower-level intermediate representation such as C--[18].

3.3 Generic Types

A generic class is implemented using a single copy of the code at run-time — it is not expanded or specialised for each use like C++ templates. Generic data is handled with a uniform representation, i.e. a pointer to the actual object. For example, the `element` field of the `Cell` class will be implemented as a pointer. The type parameter of a generic class, such as `T` in `Cell`, can abstract over all subtypes of its bound, including value classes. When a generic class is used to store value class data, the value type objects are boxed when passed into a generic context and unboxed when passed back. For example, a boxing operation will be used to implement the code fragment where the `Cell[point]` variable `c` is initialised with the value class object `p`.

To implement the lazy boxing process, the implementation of a generic type may need to consult information about the actual type of data that a type parameter has been instantiated with. To support this, we add a single member to the object instance C structure for each type parameter in the generic type. This field is akin to a vptr, but links to information about the type parameter rather than the generic type itself. Like normal vptrs, these fields are set once when the generic object is constructed.

3.4 Final Value Classes and the Vptr

The implementation of the Pluto language uses the traditional object model where an object's representation at run-time is a structure containing all the instance fields, headed with a pointer (the vptr) to some class-specific data. One important difference in the handling of value classes in Pluto is that a value class marked `final` is represented at run-time without a vptr. The rationale for this choice is that objects of a final class do not require polymorphic method calls, because the exact type of object is known at compile-time. Hence the vptr field will not be consulted and can be omitted. Since value class objects are copied by value, omitting the vptr saves space and reduces the amount of data to be copied.

Because generic code may be used to handle objects of any type, it must account for the possibility that some objects it manipulates will have a vptr and some will not. This explains the need for the extra vptr fields described above for the implementation of generic types. A type parameter vptr field is used to obtain information about the generic object, unless it indicates that objects with their own vptrs are being handled, in which case those are consulted instead. Dealing with datatypes such as integers or other final value classes that do not have vptr fields has implications for the lazy boxing optimisation — we return to this topic in Sect. 5.2.

The omission of vptrs is particularly important for performance since integers and real numbers are supported in Pluto as final value classes. Although at the Pluto language level types such as integers are declared as final value classes, these types are implemented directly using the appropriate C types rather than using a struct. For example, an integer object in Pluto is just an `int` in the generated C code, because a vptr field is unnecessary.

4 Lazy Boxing Optimisation

The brief description of the Pluto language implementation given in the previous section indicates that boxing immediately occurs whenever a value type object is passed into a generic context. This is the conventional implementation of boxing, and could be described as an *eager* strategy since it creates a heap-allocated box for the object as soon as it is passed into a generic context. However, the requirement of a uniform representation of generic values is merely that we supply a pointer to an object — the actual location of the object does not necessarily have to be on the heap. Based on this observation, we could supply a pointer to an object stored on the stack and the generic code is just as able to handle the object. The benefit of using pointers to stack-allocated objects is that we do not incur the boxing overhead of obtaining a piece of heap memory and copying the object into it.

4.1 Object Lifetimes

One difficulty with using pointers to stack-allocated objects is due to the different nature of heap and stack storage. With a heap managed by a garbage collector, an object stored on the heap exists for as long as there is a pointer to the object. With a stack, an object stored in a stack frame only exists as long as the frame exists. If generic code retains a pointer to an object that no longer exists (a so-called dangling pointer) then dereferencing that pointer could have disastrous results. To ensure the safe use of pointers we can insert a boxing operation just at the point when a stack-allocated object that is still in use is about to be destroyed. This leads to a *lazy* strategy for boxing where pointers to objects in a stack frame are used until the frame is about to be removed, whereupon any objects still required are boxed. The potential benefit of this approach is in situations where a stack-allocated object exists for at least the time that it

is used by generic code — in this case, boxing is avoided entirely. Section 5.1 describes the lazy boxing analysis used by the Pluto compiler to determine when it is safe to use pointers to stack-allocated objects.

4.2 Dynamic Boxing

In situations where a stack-allocated object does need to be boxed at a late stage, because it is still required, the boxing operation does not benefit from compile-time information about the object. In the eager approach, boxing code is inserted in the non-generic context where the type of object to be boxed is known at compile-time. A consequence of the lazy approach is that the boxing code is now in a generic context, where it only has a pointer to some stack-allocated object and needs to copy this object to the heap. The type and size of the object being pointed to is unknown at compile-time, so run-time type information must be available to guide the boxing process. We call this operation *dynamic boxing* because it works without the benefit of compile-time information. Section 5.2 explains in more detail how this is achieved in the Pluto implementation.

4.3 Example of Redundant Boxing

Naturally, the effectiveness of using the lazy strategy in general depends on the extent to which typical uses of generic code with value type objects can actually avoid boxing. One area in which redundant boxing can be found is the passing of generic data as arguments to methods. For example, the code in Fig. 1 contains a method call `c.elementAtEquals(q)` where the value type object `q` is allocated on the stack. Since the method parameter expects generic data in the uniform representation, a pointer to an object is required. Using the eager boxing strategy, the `q` object would be immediately copied to the heap and the address of this copy used as the method argument.

However, the boxing is actually redundant in this case because a pointer to `q` on the stack could have been safely used instead. We can simply pass the address of `q` (i.e. a pointer to the stack) to the `elementEquals` method, which in turn passes the pointer on as an argument to the `==` method of the generic `element` field. Since the `Cell` object is being used to contain a `Point` element type, the `==` method that is actually called is that defined by the `Point` class. Here, the required `Point` argument is obtained by dereferencing the pointer that has been passed in from the call site. This pointer refers to the `q` object on the stack, which is still in existence so the dereferencing is safe. Hence, this simple example shows one case in which use of eager boxing would result in unnecessary heap allocation and copying.

5 Implementation Details

This section describes the two main aspects to our implementation of lazy boxing: analysing when the optimisation can be made, and generating the necessary code to do dynamic boxing.

5.1 Safe Use of Stack Pointers

As briefly explained in Sect. 4.1, using pointers to objects on the stack can be problematic if the object no longer exists when a pointer to it is dereferenced. To prevent this situation from occurring, we must only generate C code that uses stack pointers in a safe manner. Our compiler adheres to the following two safety rules for the C code it generates:

1. A pointer to a stack-allocated object cannot be stored as an instance field in any object. Therefore, all instance fields that are represented with a C pointer will point to a heap-allocated object.
2. A pointer to a stack-allocated object cannot be stored in a stack frame that is older than the one in which the object is stored.

These rules ensure that a pointer is always safe, meaning it does not outlive the object that it points to. It should be noted that these are not the only possible rules for safe stack pointer usage — indeed, they can be overly conservative in some cases. Similar restrictions are made and formalised in [19], which explains how a core fragment of the Microsoft IL is type checked to prevent unsafe stack pointers.

In order to abide by these rules, the compiler must ensure that a pointer to a stack-allocated object is not used in those places where a violation could occur. Its task is to intercept stack pointers and replace them with pointers to the heap. There are two areas of the generated code that require attention:

1. Initialising or updating the value of an instance field of generic type.
2. Returning a value from a method whose declared return type is generic.

The actual replacement operation required in these situations depends on the type of value being stored or returned. If it is a value type object, then the object is eagerly boxed, which provides us with a heap pointer. However, if the value is already generic (i.e. a pointer) then the replacement task depends on whether the pointer could refer to an object on the stack or one already on the heap. If the object is on the stack then dynamic boxing is required, which is considered further in Sect. 5.2.

This section describes the situations in which unsafe uses of stack pointers could occur and shows how they are avoided. However, we are able to use pointers to stack-allocated objects in any other situation, e.g. when passing value type objects as generic value arguments. For example, Sect. 4.3 showed a case where no kind of boxing was needed. The C code generated for that fragment will just pass the address of a stack-allocated object as an argument, with no interception or replacement required.

5.2 Dynamic Boxing Using Run-Time Type Information

The main aim of lazy boxing is to use pointers to stack-allocated objects for as long as it is safe to do so. In situations where the object exists on the stack for

at least as long as any pointers to it are in use, then we avoid boxing entirely and save the cost of heap allocation and copying. However, there are cases where generic code may need to refer to an object for longer than it exists on the stack.

As explained in the previous section, if a generic pointer value is about to be stored in an object field or returned from a method then we must ensure that it points to an object on the heap. Since there is no reliable and portable way to determine in C whether a pointer actually refers to stack or heap storage[20] our compiler takes a conservative approach. We examine the actual run-time type of the object being pointed to, and if it is an instance of a reference type then we know it must already be allocated on the heap and hence no boxing is required. However, if the run-time type information indicates that the pointer refers to a value class object then our assumption is that it might be stack-allocated, so a dynamic boxing operation is needed to ensure safety. Some implications of this conservative assumption can be seen in the results shown in Sect. 6.2, which are discussed in Sect. 7.

The difference between ordinary eager boxing operations and the dynamic boxing described in this section is that the latter works in a generic context. This means that the compiler cannot determine statically the size of the object that is to be copied to the heap. The only information available in the generic context is a pointer to the object, and access to run-time type information. Consequently, a dynamic boxing operation is implemented by reading the size of the object from the run-time type information and creating a copy of the object on the heap.

From the description of the dynamic boxing process detailed here, two pieces of information about each type are required at run-time: whether the type is a value class or not, and the size of an object of that type. In our implementation, we store this information in the type-specific structure (the vtable) accessible via the `vptr` field in objects. One complication here is that objects of a final value class (such as integers or real numbers) do not have a `vptr` field, as explained in Sect. 3. In order for generic code to access run-time type information when a generic value may point to an object without a `vptr`, we maintain a pointer to the type-specific information in the generic object itself. For example, an object of the generic `Cell` class from Fig. 1 will contain a pointer to information about the particular type that it is being used for — such as `point` in the example code. This link allows the dynamic boxing code to determine the necessary information to copy an object to the heap.

6 Experimental Results

To examine the effectiveness of the lazy boxing optimisation in practice, we wrote some small test programs and built executables both with and without the boxing optimisation. Since our compiler can be configured to use either the eager or lazy boxing approaches, we were able to measure the differences in total (heap plus stack) memory usage and execution time for the test programs under the two techniques.

Table 1. Comparing reference and final value classes

Test Program	Memory Usage (Kilobytes)		Running Time (seconds)		Relative Speedup
	Reference	Final Value	Reference	Final Value	
matrix_mult	3472	1708	7.07	0.85	88%
gen_comptr_tree	704	704	3.14	2.92	7%
word_comptr_tree	704	616	3.11	2.83	9%
gen_compare	180	180	1.59	1.18	26%
word_tree	820	660	6.30	5.81	8%
gen_tree	816	1028	6.47	7.61	-18%

Since the Pluto language is an experimental one, we had no large test suites with which to measure performance. Therefore, the test programs simply exercise generic data structures such as binary search trees and matrices, where the element type is a value class. These programs are admittedly only micro-benchmarks, that perform large amounts of boxing: value type objects are stored in, and retrieved from, generic container structures. The tests were deliberately chosen to isolate the boxing activity that we are interested in optimising, so that any change in performance can be attributed to the optimisation effect. Naturally, we would need to implement the optimisation in an existing language compiler to measure its effects on larger real-world programs.

The test programs used for this work were built and run on a P3-500 PC with 384MB RAM, running Redhat Linux 6.1. We used gcc version 2.95.2 to compile the C code generated by our compiler, and statically linked the code with version 5.3 of the Boehm garbage collection library.

6.1 Measuring the Benefits of Value Types

Although our primary interest is in measuring the effect of lazy boxing, the first set of test results (shown in Table 1) just compares the performance of test programs where the element type used in a generic data structure is either a reference type or a value type. The purpose of these results is to show whether the language distinction between reference types and value types provides any benefits at all. Since the only need for any kind of boxing is due to the use of value types with genericity, it is important to establish whether a program using value type objects that are boxed as required, performs better than an equivalent program which uses reference type objects instead. If not, we could just use the uniform pointer representation for objects of all types at all times — no boxing would ever be needed.

The programs listed in Table 1 were tested in two configurations: the data structure element type was declared as either a reference class (i.e. a pointer-based representation such as that used for Java classes) or a final value class.

Table 2. Comparing eager and lazy boxing

Test Program	Memory Usage (Kilobytes)		Running Time (seconds)		Relative Speedup
	Eager Boxing	Lazy Boxing	Eager Boxing	Lazy Boxing	
matrix_mult	2036	1708	0.89	0.85	5%
gen_comptr_tree	988	704	3.58	2.92	18%
gen_compare	180	180	2.45	1.18	52%
word_tree	660	660	8.24	5.81	30%
gen_tree	816	1028	7.41	7.61	-3%
gen_stack	308	392	5.23	2.23	57%

All of the tests shown in this table were conducted with the lazy boxing optimisation enabled. The results show that, for all the programs except `gen_tree`, the amount of memory used and the running time of the final value class version was an improvement on the reference class version. Clearly, programs such as `matrix_mult`, which generates large numbers of temporary objects during its calculations, perform much better when using stack-allocated objects. However, the figures for the `gen_tree` program show that not all programs benefit from using value class objects. This particular program highlights an inefficiency in our implementation of boxing, and is examined further in the context of the boxing tests below.

The results of this set of tests show that the combination of value type objects and boxing can provide benefits over using reference type objects. While this outcome is unsurprising, and certainly not new, it provides a foundation for the following set of tests which measure the difference between different approaches to boxing.

6.2 Comparing Eager and Lazy Boxing

In the previous set of tests, we compared the difference between reference and value element types while keeping the boxing approach fixed. In the next set of tests (results shown in Table 2) the element type is fixed as a final value class, and we compare different approaches to boxing. Most of the programs used here are the same as those from the previous test set. The purpose of this experiment is to examine whether the lazy boxing optimisation leads to actual improvements in program speed and memory usage. The figures in Table 2 show that lazy boxing provides a speedup compared to eager boxing in all cases except one (the `gen_tree` program again) with a geometric mean speedup of 25% for this set of programs.

The results for the `gen_tree` program show a small slowdown and increased memory consumption for the lazy boxing version compared to eager boxing. This is counter to the trend seen in the other test programs. We investigated the cause of this unexpected result and determined that it is due to *reboxing*, where

a boxed copy of an object is made even though the original object is already on the heap. This accounts for the extra memory consumption and extra time required to make the redundant copy. We discuss the source of this inefficiency, and consider a solution, in Sect. 7 below.

7 Discussion

The initial motivation for using a lazy approach to boxing value type objects was that in some cases a boxing operation would never be needed at all. Naturally, the effectiveness of the technique depends on whether the extra complexity and possible cost of doing dynamic boxing operations at a later stage is outweighed by actual savings due to boxing being avoided completely in some cases. The results presented in the previous section certainly show that this optimisation is effective, and provides performance benefits for all but one of the test programs. As previously discussed in the review of other relevant work, there exist compile-time analysis techniques that can remove boxing operations. These could work in concert with our optimisation by removing boxing operations that are definitely redundant, leaving the remaining cases where the analysis was inconclusive to be handled using the lazy boxing approach.

Another factor in measuring the benefit of using lazy boxing is determining the range of situations in which it is most applicable and therefore effective. Since we have only tested the optimisation with small benchmark programs, it is admittedly not clear how the technique would fare with larger programs. Most of the benchmark programs involved creating value type objects and passing them into a generic context, such as code for a generic tree data structure. As stated in Sect. 4.3 there are certain patterns of code in which lazy boxing can likely be applied. For example, when passing a value type object as a generic argument to a method call, it is often the case that the pointer argument is only used to examine the object rather than store a copy of it. So the frequency of such code patterns in real programs really determines how often the lazy boxing optimisation can be exploited. A next step would be to instrument an existing language compiler that currently performs eager boxing, and measure how often lazy boxing *could* have been performed for a suite of large programs. This would tell us how widely applicable the technique is, beyond micro-benchmarks.

One design decision taken by our work (also by the C# and Eiffel languages) is that programmers should annotate classes as either reference or value type. We chose to take this approach because programmers typically have a good intuition about which data types are immutable value-like and which are shared reference-like. Adding the `value` keyword to particular class definitions is then a trivial task, so comparing program performance in the value and reference scenarios is easy to do. The alternative would be for the compiler to perform escape analysis and attempt to decide automatically which objects should be allocated on the stack and which on the heap. This is a finer-grained approach, since the allocation can be chosen per-object rather than per-class. While we did not pursue such a design, it would be useful to compare whether a programmer's

choice of value class annotation actually results in better performance than an analysis-based allocator. We believe that current language designs tend towards enabling the programmer to explicitly make the decision about which types should be value-based, hence our work focuses on implementing such languages.

The results for one of the test programs indicate that lazy boxing can perform worse than eager boxing in certain circumstances. The source of this inefficient handling can be found in the implementation of dynamic boxing as described in Sect. 5.2. Recall that the dynamic boxing process cannot determine whether a pointer actually points to the heap or stack. When the pointer refers to a value class object, it might be stack-allocated so the compiler takes a conservative approach and makes a boxed copy, to be safe. If the object had in fact already been boxed then it will be boxed again, unnecessarily. In contrast, the eager approach boxes objects exactly once. This shows a limitation of the analysis used in our implementation, which could be improved to distinguish pointers that definitely refer to objects on the heap from those that may not. Our aim is to implement the lazy boxing technique such that it never performs worse than the eager approach. Implementing this extra analysis is future work.

Although our work focuses on the boxing of value type objects for use with generic code (i.e. parametric polymorphism) the general technique of boxing may also be used to implement languages that provide polymorphism through subtyping. For example, the C# language does not currently support genericity, but it allows value type objects such as integers to be used by code that expects reference type objects. In effect, value types are considered part of the class hierarchy. The language specifies that boxing and unboxing will occur on the boundary between value and reference type contexts. Similar boxing rules appear in the Eiffel language specification. Therefore, our approach of using pointers to objects on the stack could also be applied in C# or Eiffel implementations.

8 Conclusions

We have presented an optimisation scheme for object boxing, called lazy boxing, whereby pointers to a value type object in stack storage can be used by generic code. Only when a stack-allocated object is about to be deallocated is a boxing operation necessary. This technique reduces the cost of object boxing, compared to the conventional eager approach, by avoiding boxing operations that are unnecessary. We have implemented this optimisation in a compiler for a language with value types and genericity, and showed experimental test results that demonstrate its effectiveness, achieving 25% program speedups on average. The lazy boxing optimisation scheme is applicable to language implementations that employ boxing of value type data for use in a context requiring a uniform pointer representation.

Acknowledgements. This work was financially supported by an EPSRC studentship.

References

1. Hejlsberg, A., Wiltamuth, S.: C# language reference. Microsoft Developer Network Library (2000)
2. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: Proceedings of OOPSLA'98. Volume 33(10) of SIGPLAN Notices., ACM (1998) 183–200
3. Cartwright, R., Steele, G.: Compatible genericity with run-time types for the Java programming language. In: Proceedings of OOPSLA'98. Volume 33(10) of SIGPLAN Notices., ACM (1998) 201–215
4. Odersky, M., Wadler, P.: Pizza into Java: Translating theory into practice. In: Proceedings of POPL'97, Paris, France, ACM Symposium (1997) 146–159
5. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET Common Language Runtime. In: Proceedings of PLDI'01. Volume 36(5) of SIGPLAN Notices., ACM Conference (2001) 1–12
6. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley (1994)
7. Collin, S., Colnet, D., Zendra, O.: Type inference for late binding: The SmallEiffel compiler. In: Joint Modular Languages Conference. Number 1204 in Lecture Notes in Computer Science. Springer-Verlag (1997) 67–81
8. Leroy, X.: Unboxed objects and polymorphic typing. In: Proceedings of POPL'92, Albequerque, New Mexico, ACM Symposium (1992) 177–188
9. Shao, Z., Appel, A.W.: A type-based compiler for Standard ML. In: Proceedings of PLDI'95. Volume 30(6) of SIGPLAN Notices., ACM Conference (1995) 116–129
10. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Proceedings of POPL'95, San Francisco, California, ACM Symposium (1995) 130–141
11. Shao, Z.: An overview of the FLINT/ML compiler. In: Workshop on Types in Compilation, TIC'97 (1997)
12. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: A type-directed optimizing compiler for ML. In: Proceedings of PLDI'96. Volume 31(5) of SIGPLAN Notices., ACM Conference (1996) 181–192
13. Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. ACM SIGPLAN Notices **34** (1999) 1–19
14. Gay, D., Steensgaard, B.: Fast escape analysis and stack allocation for object-based programs. In: International Conference on Compiler Construction (CC'2000). Volume 1781., Springer-Verlag (2000)
15. Owen, T.: A Type-Passing Implementation of Value Types and Genericity. PhD thesis, University of Sussex, Brighton, UK (2002) Available at <http://www.cogs.susx.ac.uk/users/timo/thesis.pdf>.
16. Meyer, B.: Eiffel: The Language. Prentice-Hall (1992)
17. Boehm, H.J.: Space efficient conservative garbage collection. ACM SIGPLAN Notices **28** (1993) 197–206
18. Peyton Jones, S.L., Nordin, T., Oliva, D.: C--: A portable assembly language. In: Proceedings of the 1997 Workshop on Implementing Functional Languages. Lecture Notes in Computer Science. Springer-Verlag (1998)
19. Gordon, A., Syme, D.: Typing a multi-language intermediate code. In: Proceedings of POPL 2001, London, England, ACM Symposium (2001) 248–260
20. Meyers, S.: More Effective C++. Addison-Wesley (1996)

FFT Compiler Techniques

Stefan Kral, Franz Franchetti, Juergen Lorenz, Christoph W. Ueberhuber, and Peter Wurzinger

Institute for Analysis and Scientific Computing,
Vienna University of Technology,
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
`skral@complang.tuwien.ac.at`,
`www.math.tuwien.ac.at/ascot`

Abstract. This paper presents compiler technology that targets general purpose microprocessors augmented with SIMD execution units for exploiting data level parallelism. Numerical applications are accelerated by automatically vectorizing blocks of straight line code to be run on processors featuring two-way short vector SIMD extensions like Intel's SSE 2 on Pentium 4, SSE 3 on Intel Prescott, AMD's 3DNow!, and IBM's SIMD operations implemented on the new processors of the BlueGene/L supercomputer.

The paper introduces a special compiler backend for Intel P4's SSE 2 and AMD's 3DNow! which is able (*i*) to exploit particular properties of FFT code, (*ii*) to generate optimized address computation, and (*iii*) to perform specialized register allocation and instruction scheduling.

Experiments show that the automatic SIMD vectorization techniques of this paper enable performance of hand optimized code for key benchmarks. The newly developed methods have been integrated into the codelet generator of FFTW and successfully vectorized complicated code like real-to-halfcomplex non-power of two FFT kernels. The floating-point performance of FFTW's scalar version has been more than doubled, resulting in the fastest FFT implementation to date.

1 Introduction

Major vendors of general purpose microprocessors have included short vector single instruction multiple data (SIMD) extensions into their instruction set architecture to improve the performance of multimedia applications by exploiting data level parallelism. The newly introduced instructions have the potential for outstanding speed-up, but they are difficult to utilize using standard algorithms and general purpose compilers.

Recently, a new software paradigm emerged in which optimized code for numerical computation is generated automatically [4,9]. For example, FFTW has become the de facto standard for high performance FFT computation. The current version of FFTW includes the techniques presented in this paper to utilize SIMD extensions.

This paper presents compiler technology for automatically vectorizing numerical computation blocks as generated by automatic performance tuning systems like FFTW, SPIRAL, and ATLAS. The blocks to be vectorized may contain load and store operations, index computation, as well as arithmetic operations.

In particular, the paper introduces a special compiler backend which generates assembly code optimized for short vector SIMD hardware. This backend is able to exploit special features of straight-line FFT code, generates optimized address computation, and applies additional performance boosting optimization.

The newly developed methods have been integrated into FFTW's codelet generator yielding FFTW-GEL [6], a short vector SIMD version of FFTW featuring outstanding floating-point performance (see Section 5).

Related Work. Established vectorization techniques mainly focus on loop-constructs. For instance, Intel's C++ compiler and Codeplay's VECTOR C compiler are able to vectorize loop code for both integer and floating-point short vector extensions.

An upgrading to the SUIF compiler that vectorizes loop code for MMX is described in [11].

A SPIRAL based approach to portably vectorize discrete linear transforms utilizing structural knowledge is presented in [2].

A vectorizing compiler exploiting *superword level parallelism* (i.e., SIMD style parallelism) has been introduced in [8]. IBM's XL C compiler for Blue-Gene/L [13] utilizes this vectorization technique.

Intel's math kernel library (MKL) and performance primitives (IPP) both support SSE and SSE 2 available on Pentium III and 4 processors and the Itanium processor family, as well as the new SSE 3 on Intel Prescott.

A backend for straight-line code targeting MIPS processors is presented in [5].

2 FFTW for Short Vector Extensions: FFTW-GEL

FFTW-GEL¹ is an extended version of FFTW that supports two-way short vector SIMD extensions (see Fig. 1).

FFTW is an automatic FFT code generator based on a recursive implementation of the Cooley-Tukey FFT algorithm. FFTW uses dynamic programming with measured run times as cost function to find a fast implementation for a given problem size on a given machine.

FFTW consists of the following parts: *Planner* and *executor* provide for the adaptation of the FFT computation to the target machine at runtime while the actual computation is done within routines called *codelets*. Codelets are generated by the *codelet generator* `genfft`, a special purpose compiler.

Within the codelets a variety of FFT algorithms is used, including the Cooley-Tukey algorithm, the split radix algorithm, the prime factor algorithm, and the Rader algorithm.

¹ available from <http://www.fftw.org/~skral>

The compiler techniques presented in this paper were used to extend FFTW’s codelet generator to produce vectorized codelets. These vectorized codelets are compatible with FFTW’s framework and thus can be used instead of the original codelets on machines featuring two-way short vector SIMD extensions. That way FFTW’s core routines are sped up while all features of standard FFTW remain supported.

Short Vector SIMD Extensions. Examples of two-way short vector SIMD extensions supporting both integer and floating-point operations include Intel’s streaming SIMD extensions SSE 2 and SSE 3, AMD’s 3DNow! family, as well as IBM’s PowerPC 440d processors in BlueGene/L supercomputers.

Double-precision short vector SIMD extensions paved the way to high performance scientific computing. However, special code is needed as conventional scalar code running on machines featuring these extensions utilizes only a small fraction of the potential performance.

Short vector SIMD extensions are advanced architectural features which are not easily utilized for producing high performance codes. Currently, two approaches are commonly used to utilize SIMD instructions.

Vectorizing Compilers. The application of compiler vectorization is restricted to loop-level parallelism and requires special loop structures and data alignment. Whenever the compiler cannot prove that a loop can be vectorized optimally using short vector SIMD instructions, it has to resort to either emitting scalar code or to introducing additional code to handle special cases.

Hand-Coding. Compiler vendors provide extensions to the C language (data types and intrinsic or built-in interfaces) to enable the direct use of short vector SIMD extensions from within C programs. Of course, assembly level hand-coding is unavoidable if there is no compiler support.

2.1 FFTW-GEL

Both vectorizing compilers and hand-coding are not directly applicable to generate vectorized FFTW codelets to replace the standard codelets. Due to its internal structure, FFTW cannot be vectorized using compilers focusing on loop vectorization. FFTW automatically generates code consisting of up to thousands of lines of numerical straight-line code. Accordingly, such basic blocks have to be vectorized rather than loops.

Besides, large codelets are favored over smaller codelets used in loop bodies in the creation of FFTW plans as they feature less reorder operations. Therefore, the automatic vectorization of large basic blocks is an important task. Vectorizing them by hand is generally unfeasible as the instruction counts in Fig. 2 illustrate.

FFTW-GEL comprises an extended, architecture-specific version of FFTW’s `genfft`, supporting two-way short vector SIMD extensions. Moreover, it includes a special compiler backend for x86 architectures which generates assembly code optimized for short vector SIMD hardware. This backend is able to exploit special features of straight-line FFT code, generates optimized address computation, and applies additional performance boosting optimization.

FFTW-GEL’s Architecture. Fig. 1 illustrates the various levels of optimization performed by FFTW-GEL.

Vectorization of Scalar Straight-Line Code. Within the context of FFTW, codelets do the bulk of the computation. The challenge addressed by the vectorizer is to extract parallelism out of sequences of scalar instructions while maintaining data locality and utilizing special features of 2-way short vector SIMD extensions.

Optimization of Vectorized Code. Optimizer I comprises a local rewriting system using a set of rules to optimize the DAG of SIMD instructions obtained from the vectorizer.

Assembly Backend for Straight-Line Code. The backend uses (i) efficient methods to compute effective addresses, (ii) a register allocator, (iii) an instruction scheduler, (iv) an optimizer for in-memory operands, (v) a register reallocator, and (vi) address generation interlock (AGI) prevention for code size reduction.

The register reallocator, the direct use of in-memory operands, and other optimization techniques and methods, are executed in a feedback driven optimization loop to further improve the optimization effect.

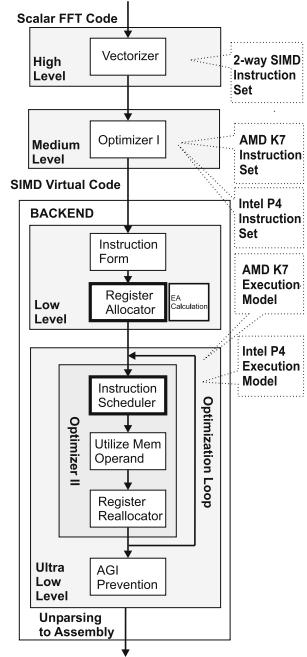


Fig. 1. FFTW-GEL’s Architecture. Automatically generated FFT DAGs are vectorized and optimized.

3 Vectorization of Scalar Straight-Line Code

This section introduces FFTW-GEL [7], a vectorizer that automatically extracts *2-way SIMD parallelism* out of a sequence of operations from static single assignment (SSA) straight-line code while maintaining data locality and utilizing special features of short vector SIMD extensions.

¹ FFTW-GEL produces vectorized code either (*i*) via a source-to-source transformation delivering macros compliant with the portable SIMD API [14] and additionally providing support for FMA instructions, or (*ii*) via a source-to-assembly transformation utilizing the FFTW-GEL backend of Section 5.

3.1 The Vectorization Approach

The vectorizer aims at vectorizing straight-line code containing arithmetic operations, array access operations and index computations. Such codes cannot be handled by established vectorizing compilers mainly focussing on loop vectorization [15]. Even standard methods for vectorizing straight-line code [8] fail in

producing high performance vector code as well. FFTW-GEL’s vectorizer uses a backtracking search engine to automatically extract *2-way SIMD parallelism* out of scalar code blocks by fusing pairs of scalar temporary variables into SIMD variables, and by replacing the corresponding scalar instructions by vector instructions as illustrated by Table 1.

Tuples of scalar variables, i. e., *fusions*, are declared such that every scalar variable appears in exactly one of them. Each fusion is assigned one SIMD variable.

On the set of SIMD variables, SIMD instructions have to perform exactly the same computation as the originally given scalar instructions do on the scalar variables. This is achieved by rules specifying how each pair of scalar instructions is replaced by a sequence of semantically identical SIMD instructions operating on the corresponding SIMD variables.

Rules for pairs of scalar binary instructions allow up to three different, semantically equivalent ways to do so. Locally, this allows vectorizations of different efficiency. Globally, it widens the search space of the vectorizer’s backtracking search engine.

The vectorizer’s goal is to yield an optimal utilization of SIMD resources. This target and the fact that the backtracking engine uses a *pick first result* strategy to minimize vectorization runtime does not necessarily lead to the best possible vectorization in terms of an overall minimum of SIMD instructions but allows the vectorization of large FFTW codelets (see Table 2).

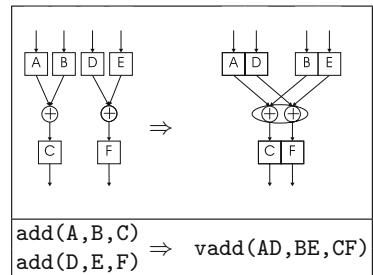
FFTW-GEL’s vectorizer utilizes an automatic fallback to different vectorization levels in its vectorization process. Each level provides a set of pairing rules with different pairing restrictions. Thus, more restrictive rules are just applied to highly parallel codes resulting either in efficient vectorization or no vectorization at all. Less restrictive rules address a broader range of codes but result in less efficient vectorization.

This concept assists the searching for good vectorizations needing a minimum of data reorganization, provided such vectorizations exist. The fallback to a less restrictive vectorization level allows to vectorize even codes featuring less parallel parts to some extent.

Virtual Machine Models. To keep the amount of hardware specific details needed in the vectorization as small as possible, *virtual machine models* are introduced.

The virtual machine models used in the vectorizer are abstractions of scalar as well as 2-way SIMD architectures, i. e., the vectorizer transforms virtual scalar to virtual 2-way SIMD instructions. These are rewritten into 2-way SIMD in-

Table 1. 2-way Vectorization. Two scalar `add` instructions are transformed into a vector `vadd` instruction. The result of the vector addition of the fusions AD and BE is stored in CF.



structions actually available in a specific architecture’s instruction set in an optimization process, directly following the vectorization.

3.2 The Vectorization Engine

The input of the vectorization engine is a scalar DAG representing straight-line code of virtual scalar instructions in SSA form. In the vectorization engine all virtual scalar instructions are replaced by virtual SIMD instructions.

The vectorization algorithm described in the next section utilizes the infrastructure provided by FFTW-GEL’s vectorization engine. To describe the concepts of the vectorization engine, some definitions and explanations are provided.

Pairing. *Pairing rules* specify ways of transforming pairs of scalar instructions into a single SIMD instruction or a sequence of semantically equivalent SIMD instructions. A pairing rule often provides several alternatives to do so. The rules used in the vectorizer are classified according to the type of the scalar instruction pair: unary (i.e., multiplication by a constant), binary (i.e., addition and subtraction), and memory access type (i.e., load and store instructions). Pairings of the following instruction combinations are supported: (i) consecutive load/load, (ii) arbitrary load/load, (iii) consecutive store/store, (iv) arbitrary store/store, (v) unary/unary, (vi) binary/binary, (vii) unary/binary, (viii) unary/load, and (ix) load/binary. Not all of the above pairing combination allow for an optimal utilization of SIMD resources.

A *pairing ruleset* comprises various pairing rules. At the moment, the pairing ruleset does not comprise a rule for vectorizing an odd number of scalar store instructions.

Two scalar instructions are vectorized, i.e., *paired*, if and only if neither of them is already paired and the instruction types are matching a pairing rule from the utilized pairing rule set.

Fusion. Two scalar operands S and T are assigned together, i.e., *fused*, to form a SIMD variable of layout $ST = (S, T)$ or $TS = (T, S)$ if and only if they are the corresponding operands of instructions considered for pairing and neither of them is already involved in another fusion. The position of the scalar variables S and T inside a SIMD variable (either as its lower or its higher part) strictly defines the fusion, i.e., $ST \neq TS$. A special fusion, i.e., $TT = (T, T)$, is needed for some types of code partly containing non-parallel program flow.

An instruction pairing rule forces the fusion layout for the corresponding scalar operands. For two scalar binary instructions, three substantially different layouts for assigning the four operands to two SIMD variables exist, namely (i) accumulate (ii) parallel 1, and (iii) parallel 2.

A fusion $X_{12} = (X_1, X_2)$ is *compatible* to another fusion $Y_{12} = (Y_1, Y_2)$ if and only if $X_{12} = Y_{12}$ or $X_1 = Y_2$ and $X_2 = Y_1$. In the second case, a swap operation is required as an additional “special transformation” to use Y_{12} whenever X_{12} is needed. The number of special transformations is minimized in a separate optimization step to minimize the runtime of the vectorization process. Section 4 describes this along other optimization techniques in more detail.

Vectorization Levels. A vectorization level embodies a subset of rules from the overall set of pairing rules. Different subsets belonging to different levels comprise pairing rules of different versatility. Generally, more versatile rule sets lead to less efficient vectorization but allow to operate on codes featuring less parallelism. In contrast, less versatile rule sets are more restrictive in their application and are thus only usable for highly parallel codes. They assure that if a good solution exists at all, it is found resulting in highly performant code.

The vectorization engine utilizes three levels of vectorization in its search process as follows: First, a vectorization is sought that utilizes the most restrictive level, i. e., *full vectorization*. This vectorization level only provides pairing rules for instructions of the same type and allows quadword memory access operations exclusively. If full vectorization is not obtainable, a fallback to a less restrictive vectorization level, i. e., *semi vectorization*, is made. This level provides versatile pairing rules for instructions of mixed type and allows doubleword memory access operations. In the worst case, if neither semi vectorization nor full vectorization is feasible, a fallback is made to a vector implementation of scalar code, i. e., *null vectorization* is applied. Null vectorization rules allow to vectorize any code by leaving half of each SIMD instruction's capacity unused. Even null vectorization results in better performance than using legacy x87 code. Besides, null vectorization is used as default, without trying full and semi vectorization, when it is known in advance that vectorization is impossible because, e. g., of an odd number of scalar store instructions, etc.

3.3 The Vectorization Algorithm

FFTW-GEL's vectorization algorithm implements a depth first search with chronological backtracking. The search space is given by applying rules given by the current vectorization level in an arbitrary order. Depending on how versatile/restrictive the utilized pairing rule set is, there can be many, one or no possible solution at all.

The vectorization algorithm performs the following steps:

Step 1: Initially, no scalar variables are fused and no scalar instructions are paired. The process starts by pairing two arbitrary store instructions and fusing the corresponding source operands. Should the algorithm backtrack without success, it tries possible pairings of store instructions, one after the other.

Step 2: Pick an existing fusion on the vectorization path currently being processed, whose two writing instructions have not yet been paired. As the scalar code is assumed to be in SSA form, there is exactly one instruction that uses each of the fusion's scalar variables as its destination operand. According to the vectorization level and the type of these instructions, an applicable pairing rule is chosen. If all existing fusions have already been processed, i. e., the dependency path has been successfully vectorized from the stores to all affected loads, start the vectorization of another dependency path by choosing two remaining stores. If all stores have been paired and no fusions are left to be processed, a solution has been found and the algorithm terminates.

Step 3: According to the chosen pairing rule, fuse the source operands of the scalar instructions if possible (i. e., none of them is already part of another fusion) or, if a compatible fusion exists use it instead.

Step 4: Pair the chosen instructions, i. e., substitute them by one or more according SIMD instructions.

Step 5: If a fusion or pairing alternative does not lead to a valid vectorization, choose another pairing rule. If none of the applicable rules leads to a solution, fail and backtrack to the most recent vectorization step.

Steps 2 to 5 are iterated until all scalar instructions are vectorized, possibly requiring new initialization carried out by Step 1 during the search process. If the search process terminates without having found a result, a fallback to the next more general vectorization level is tried, leading to null vectorization in the worst case.

If a given rule set is capable of delivering more than one valid solution, the order in which the pairing rules are tested is relevant for the result. This is used to favor specific kinds of instruction sequences by ranking the corresponding rules before the others. For instance, the vectorization engine is forced first to look for instructions implementing operation semantics directly supported by a given architecture, thus minimizing the number of extracted virtual instructions that have to be rewritten in the optimization step.

Table 2 shows runtimes of the vectorization algorithm for some representative FFTW codelets. Even large codelet, e. g., $n = 256$, can be vectorized in a few seconds.

Table 2. Vectorization Runtimes. The table shows instruction counts as well as vectorization runtimes in seconds for various twiddle complex-to-complex, no-twiddle complex-to-complex, and real-to-halfcomplex FFTW codelets. The vectorization algorithm has been implemented in Objective Caml. The runtimes have been determined using Objective Caml v3.06 with native-code compilation on an 800 MHz AMD K7 processor.

Code Name	Loads/Stores	Adds+Subs	Muls	Vectorization Runtime
frc_32	32/32	156	42	<.1s
frc_30	30/30	158	56	5.6s
ftw_17	66/34	328	180	<.1s
fn_128	256/256	2164	660	0.7s
fn_256	512/512	5008	1656	2.2s

4 Optimization of Vectorized Code

After the vectorizer terminates successfully by delivering a vectorization of the scalar DAG, Optimizer I performs several simple improvements to the resulting

code. For that purpose, a rewriting system is used to simplify combinations of virtual SIMD instructions, also with regard to target architecture specifications.

The first group of rewriting rules that is applied aims at (*i*) minimizing the number of instructions, (*ii*) eliminating redundancies and dead code, (*iii*) reducing the number of source operands (copy propagation), and (*iv*) performing constant folding. The critical path length of the DAG is shortened by exploiting specific properties of the target instruction set. Finally, the number of source operands necessary to perform an operation is reduced, thus minimizing register pressure.

The second group of rules is used to rewrite virtual instructions into combinations of (still virtual) instructions actually supported by the target architecture.

In a third and last step the Optimizer I schedules and topologically sorts the instructions of the vectorized DAG. The scheduling algorithm minimizes the lifespan of variables by improving the locality of variable accesses. It is an extension of FFTW 2.1.3's scheduler.

The code output by Optimizer I consists of instructions out of a subset of the virtual SIMD instruction set that corresponds to the target architecture.

5 Assembly Backend for Straight-Line Code

The FFTW-GEL backend [7] introduced in this section generates assembly code optimized for short vector SIMD hardware. It is able to exploit special features of automatically generated straight-line codes. The backend currently supports assembly code for x86 with 3DNow! or SSE2.

The automatically generated codes to be translated are large blocks of SSA code with the following properties: (*i*) there is no program control flow except the basic block's single point of entry and exit, (*ii*) there is full knowledge of future temporary variable usage, (*iii*) there are indexed memory accesses possibly with a stride as runtime parameter, and (*iv*) loads from and stores to data vector elements are performed only once.

Standard backends fail to compile such blocks of SSA code to performant machine code as they are targeted at a broad range of structurally different hand-written codes lacking the necessary domain specific meta information mentioned above. Thus, standard backends fail in register allocation when too many temporary variables are to be assigned to a small number of registers and they also have trouble with efficient calculation of effective addresses.

5.1 Optimization Techniques Used In FFTW-GEL

The FFTW-GEL backend performs optimization in both an *architecture specific* and a *processor specific* way.

In particular, (*i*) register allocation, (*ii*) computation of effective addresses, (*iii*) usage of in-memory operands, and (*iv*) the register reallocator are optimizations with respect to an architecture's instruction set. (*v*) Instruction scheduling

and (vi) AGI prevention is done specifically for the target processor in the last optimization step.

Architecture specific optimization takes into account very general properties of the target processor family. These are (i) the number of available physical registers, and (ii) instruction set specific properties, e.g., x86 instruction forms, and special x86 instructions, e.g., `lea`.

Processor families, all of whose members support one and the same instruction set, may still have different instruction latencies and throughput. Therefore, processor specific optimization has to take the execution properties of any specific processor into account. These properties are provided by processor-specific execution models specifying (i) how many instructions can be issued per clock cycle, (ii) the latency of each instruction, (iii) the execution resources required by each instruction, and (iv) the overall available resources.

5.2 One-Time Optimization

Register allocation, computation of effective addresses, and avoidance of address generation interlocks are optimization techniques performed only once.

Register Allocation for Straight-Line Code. The register allocator's input code contains vector computation as well as its integer address computation accessing input and output arrays in memory. Thus, registers have to be allocated for both register types assuming two different target register files.

The codes are in SSA form and thus only one textual definition of a variable exists. There is no control flow either. As a consequence of these properties, it is possible to evaluate the effective live span of each temporary variable, and thus, a farthest first algorithm [12] can be used as a spilling scheme that tries to find a *spill victim* in the following order: (i) take a fresh physical register if available, (ii) choose among the dead registers the one which has been dead for the longest time, and finally if no register of any other kind is available, (iii) choose the register whose usage is ahead furthermost.

General purpose compilers cannot apply the farthest first algorithm in their spilling schemes as they also address codes possibly containing complex control flow structures not allowing to precisely determine the life span of each temporary variable.

Optimized Index Computation. FFTW codelets operate on arrays of input and output data not necessarily stored contiguously in memory. Thus, access to element `in[i]` may result in a memory access at address `in + i*sizeof(in)*stride`. Both `in` and `stride` are parameters passed from the calling function.

All integer instructions, except the few ones used for parameter passing, are used for calculating effective addresses. For a codelet of size N , the elements `in[i]` with $i = 0, 1, \dots, N - 1$ are accessed exactly once. Still, the quality of code dealing with address computation is crucial for achieving a satisfactory performance.

Instead of using general integer multiplication instructions (`imull`, etc.), it has been demonstrated to be advantageous to utilize (*i*) equivalent sequences of simpler instructions (`add`, `sub`, `shift`) which can be decoded faster, have shorter latencies, and are all fully pipelined, and (*ii*) instructions with implicit integer computation, like the `lea` instruction. Moreover, (*iii*) reusing the content of integer registers using the integer register file is beneficial to avoid costly integer multiplications as far as possible.

The Load Effective Address Instruction. The generation of code sequences for calculating effective addresses is intertwined with the register allocation process. On x86 compatible hardware architectures, the basic idea to achieve an efficient computation of effective array addresses is to use the powerful `lea` instruction which combines up to two adds and one shift operation in one instruction: `base + index*scale + displacement`. This operation can be used to quickly calculate addresses of array elements and therefore is a cheap alternative to general multiplication instructions.

Whenever the register allocator needs to emit code for the calculation of an effective address, the (locally) shortest possible sequence of `lea` instructions is determined by depth-first iterative deepening (DFID) with a target architecture specific depth limit (e.g., 3 for the AMD K7). As the shortest sequences tend to eagerly reuse already calculated contents of the integer register file, a replacement policy based on the least recently used (LRU) heuristic is employed for integer registers.

The generation of code sequences for calculating effective addresses is intertwined with the register allocation process.

*Example 1 (Efficient Computation of 17*stride).* The `lea` instruction is used to compute `17*stride` from the current entries of the integer register file. In this example all required operands already reside in the register file providing three different ways of computing the result. The entries in the register file and the respective `lea` instructions are displayed below.

Integer Register File				LEA Instructions	Result
<code>eax</code>	<code>stride*3</code>	<code>ebx</code>	<code>stride*5</code>	<code>lea ecx, [ebx + 4*eax]</code>	$(5 + 4 * 3) * \text{stride}$
<code>edx</code>	<code>stride*1</code>	<code>esp</code>	<code>stride*4</code>	<code>lea ecx, [edi + 2*esi]</code>	$(-1 + 2 * 9) * \text{stride}$
<code>esi</code>	<code>stride*9</code>	<code>edi</code>	<code>stride*-1</code>	<code>lea ecx, [edx + 4*esp]</code>	$(1 + 4 * 4) * \text{stride}$

Avoiding Address Generation Interlocks. Memory access operations do not have WAW dependencies in the straight-line FFT codes of this paper as writing to a specific memory location happens only once. Thus, any order of memory instructions obeys the given program semantics. This enables the FFTW-GEL backend to reorder store instructions to resolve address generation interlocks (AGIs). AGIs occur whenever a memory operation involving some expensive effective address calculation directly precedes a memory operation involving inexpensive address calculation or no address calculation at all. Generally, as there is no knowledge whether there are WAW dependencies between memory accessing

instructions or not in the codes, the second access is stalled by the computation of the first access' effective address. Example 2 illustrates the prevention of an AGI.

Example 2 (Address Generation Interlock). A write operation to `out[17]` is directly preceding a write operation to `out[0]` in the assumed code. As hardware requires an in-order memory access, writing to `out[0]` requires a completed writing to `out[17]` and thus the computation of the effective address of `out[17]`. Both would stall write to `out[0]`. By knowing that there are no WAW dependencies between `out[0]` and `out[17]`, these operations are swapped. The write to `out[0]` is done *concurrently* to computing the effective address of `out[17]` and thus its writing is not stalled.

```
addr = 17 * stride;      addr = 17 * stride;
out[addr] = temp0;      → out[0] = temp1;
out[0] = temp1;         out[addr] = temp1;
```

AGI prevention is performed as FFTW-GEL's last optimization after the feedback driven optimization loop.

5.3 Feedback Driven Optimization

Instruction scheduling, direct use of in-memory operands, and register reallocation are executed in a feedback driven optimization loop to further improve the optimization effect. The instruction scheduler serves as a basis for estimating the runtime of the entire basic block. As long as the code's estimated execution time can be improved, the feedback driven optimizations are executed.

Basic Block Instruction Scheduling. The instruction scheduler of the FFTW-GEL backend aims at single basic blocks. It uses a standard list scheduling algorithm [10,12]. The processor's execution behavior (latencies, etc.) and resources (issue slots, execution units, ports) are taken into account. These properties are provided by externally specified processor execution models, currently available for Intel's Pentium 4 and AMD's K6 and K7.

Direct Usage Of In-Memory Operands. The FFTW-GEL backend enables the usage of an in-memory operand instead of registers for an instruction's source operand. This is possible as many CISC instruction sets, e.g., x86, allow one source operand of an instruction to reside in memory. It enables a reduction of register pressure and a reduction of the total number of instructions by merging one load and one use instruction into one load-and-use instruction.

FFTW-GEL directly applies in-memory operands if and only if the data is accessed by only one instruction. This prevents register files from not being fully utilized and the superfluous loading of data.

This optimization is performed intertwined with the instruction scheduler and a postprocessing register reallocator.

Register Reallocation. Register allocation and instruction scheduling have conflicting goals. As the register allocator tries to minimize the number of register spills, it prefers introducing a false dependency on a dead logical register

over spilling a live logical register. The instruction scheduler, however, aims at the maximization of the pipeline usage of the processor, by spreading out the dependent parts of code sequences according to the latencies of the respective instructions.

As FFTW-GEL performs register allocation before doing instruction scheduling, it is clear that the false dependencies introduced by the register allocator severely reduce the number of possible choices of the instruction scheduler.

To address this problem, the register reallocator tries to lift some (too restrictive) data dependencies introduced by the register allocator (or by a previous pass of register reallocation), enabling a following pass of the instruction scheduler to do a better job. The register reallocator renames all logical SIMD registers using a least-recently-used (LRU) heuristic.

6 Experimental Results

Numerical experiments were carried out to demonstrate the applicability and the performance boosting effects of the newly developed compiler techniques [7].

FFTW-GEL was investigated on two IA-32 compatible machines: (i) An Intel Pentium 4 featuring SSE 2 two-way double-precision SIMD extensions [7], and (ii) an AMD Athlon XP featuring 3DNow! professional two-way single-precision SIMD extensions (see Fig. 2).

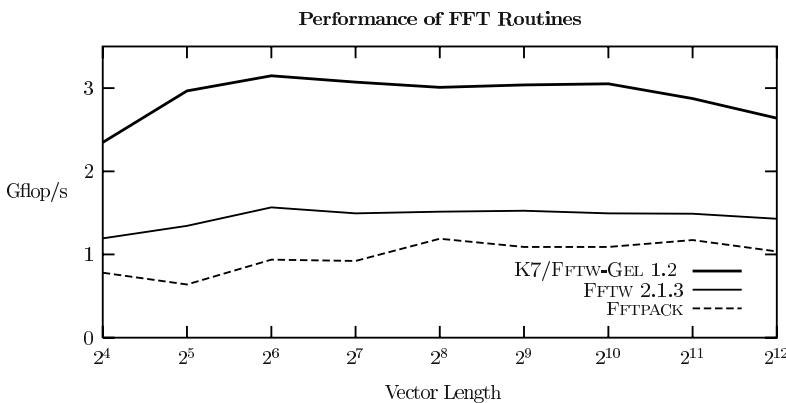


Fig. 2. Floating-point performance of the newly developed K7/FFTW-GEL (3DNow!) compared to FFTPACK and FFTW 2.1.3 on a 1.53 GHz AMD Athlon XP 1800+ carrying out complex-to-complex FFTs in single-precision. Performance data are displayed in pseudo-Gflop/s, i. e., $5N \log N/T$.

Fig. 2 shows the performance of FFTPACK, FFTW 2.1.3, and K7/FFTW-GEL on the Athlon XP in single-precision. The runtimes displayed refer to powers of two complex-to-complex FFTs whose data sets fit into L2 cache. The newly developed K7/FFTW-GEL utilizes the enhanced 3DNow! extensions which provide two-way single-precision SIMD operations. FFTW-GEL is about twice as

fast as FFTW 2.1.3, which demonstrates that the performance boosting effect of vectorization and backend optimization is outstanding.

Very recently, experiments were carried out on a prototype of IBM's BlueGene/L (BG/L) top performance supercomputer. Fig. 3 shows the relative performance of FFTW 2.5.1 no-twiddle codelets.

IBM's XLC compiler for BlueGene/L using code generation *with* SIMD vectorization and *with* FMA extraction (using the compiler techniques [8]) sometimes accelerates the code slightly but also slows down the code in some cases. FFTW-GEL's vectorization yields speed-up values up to 1.8 for sizes where the XLC compiler's register allocator generates reasonable code. For codes with more than 1000 lines (size 16, 32, 64) the performance degrades because of the lack of a good register allocation.

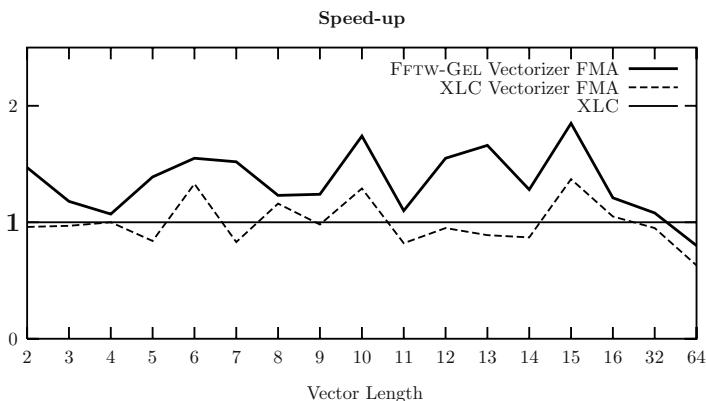


Fig. 3. Speed-up of (i) the newly developed BlueGene/L FFTW-GEL vectorizer *with* FMA extraction (but without backend optimizations) and (ii) FFTW codelets vectorized by XLC *with* FMA extraction compared to (iii) scalar FFTW codelets using XLC *without* FMAs, and *without* vectorization. The experiment has been carried out running no-twiddle codelets.

Conclusion

This paper presents a set of compilation techniques for automatically vectorizing numerical straight-line code. As straight-line code is in the center of all current numerical performance tuning software, the newly developed techniques are of particular importance in scientific computing.

Impressive performance results demonstrate the usefulness of the newly developed techniques which can even vectorize the complicated code of real-to-halfcomplex FFTs for non-powers of two.

Acknowledgement. We would like to thank Matteo Frigo and Steven Johnson for many years of prospering cooperation and for making it possible for us to access non-public versions of FFTW.

Special thanks to Manish Gupta, José Moreira, and their group at IBM T.J. Watson Research Center (Yorktown Heights, N.Y.) for making it possible to work on the BG/L prototype and for a very pleasant and fruitful cooperation.

The Center for Applied Scientific Computing at LLNL deserves particular appreciation for ongoing support.

Additionally, we would like to acknowledge the financial support of the Austrian science fund FWF.

References

1. Belady, L.A.: A study of replacement algorithms for virtual storage computers. *IBM Systems Journal* **5** (1966) 78–101
2. Franchetti, F., Püschel, M.: A SIMD vectorizing compiler for digital signal processing algorithms. In: Proc. International Parallel and Distributed Processing Symposium (IPDPS'02). (2002)
3. Frigo, M.: A fast Fourier transform compiler. In: Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation. ACM Press, New York (1999) 169–180
4. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: ICASSP 98. Volume 3. (1998) 1381–1384
5. Guo, J., Garzarán, M.J., Padua, D.: The Power of Belady's Algorithm in Register Allocation for Long Basic Blocks. In: Proceedings of the LCPC 2003
6. Kral, S., Franchetti, F., Lorenz, J., Ueberhuber, C.W.: SIMD vectorization techniques for straight line code. Technical Report AURORA TR2003-02, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology (2003)
7. Kral, S., Franchetti, F., Lorenz, J., Ueberhuber, C.W.: SIMD Vectorization of Straight Line FFT Code. In: Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790, Springer-Verlag, Berlin (2003) 251–260
8. Larsen, S., Amarasinghe, S.: Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices* Vol. **35** 5 (2000) 145–156
9. Moura, J.M.F., Johnson, J., Johnson, R.W., Padua, D., Prasanna, V., Püschel, M., Veloso, M.M.: SPIRAL: Portable Library of Optimized Signal Processing Algorithms (1998). <http://www.ece.cmu.edu/spiral>
10. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufman Publishers, San Francisco (1997)
11. Sreraman, N., Govindarajan, R.: A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming* **28** (2000) 363–400
12. Srikant, Y.N., and Shankar, P.: The Compiler Design Handbook. CRC Press LLC, Boca Raton London New York Washington D.C. (2003)
13. G. Almasi et al., “An overview of the BlueGene/L system software organization,” Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing LNCS 2790, 2003.
14. F. Franchetti and C. Ueberhuber, “An abstraction layer for SIMD extensions,” Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, Technical Report AURORA TR2003-06, 2003.
15. Intel Corporation, “Intel C/C++ compiler user's guide,” 2002.

Widening Integer Arithmetic

Kevin Redwine and Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University
`{redwine,nr}@eecs.harvard.edu`

Abstract. Some codes require computations to use fewer bits of precision than are normal for the target machine. For example, Java requires 32-bit arithmetic even on a 64-bit target. To run narrow codes on a wide target machine, we present a *widening* transformation. Almost every narrow operation can be widened by sign- or zero-extending the operands and using a target-machine instruction at its natural width. But extensions can sometimes be avoided, and our transformation avoids as many as possible. The key idea is knowing what each operation can accept in the high bits of its arguments and what it can guarantee about the high bits of its result. This knowledge is formalized using *fill types*, which drive the widening transformation.

1 Introduction

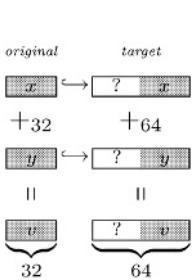


Fig. 1. Widening 32-bit addition:
 $x +_{32} y$

Machines can perform arithmetic and logical operations only on values of a particular width, often 32 or 64 bits. A compiler that knows the width of its target machine can generate code containing only operations of the correct width. But sometimes a different width is required by the source code. For example, Java requires 32-bit operations, but a compiler for IA-64 must generate 64-bit operations. Or if the source is binary code, as in a binary translator, then the width of operations is determined by the source machine. In general, if the source code requires widths not available on the target machine, a translator must change the widths without changing the meaning of the program. For example, a translated program might simulate 32-bit operations using 64-bit hardware. We call the relevant translation *widening*.

Take, for example, a program that adds two 32-bit values and produces a 32-bit result. A simple way to simulate 32-bit addition on a 64-bit machine is to widen the operand values to 64 bits and perform a 64-bit addition. The result of the original 32-bit addition is in the least significant, or low, 32 bits of the 64-bit result. This example is illustrated in Fig. 1. The heart of the widening problem is to determine how to fill in the high bits, which are shown as question marks in Fig. 1.

A simple solution is to explicitly sign extend (or zero extend) every variable and every intermediate result, ensuring that the wide and narrow representations have exactly the same denotation as a two's-complement (respectively unsigned) integer. But for many expressions, this would be overkill; for example, in the addition above, the correct value v appears in the low 32 bits of the result regardless of what is in the high bits of the operands. A good solution, therefore, minimizes the number of explicit sign extensions or zero extensions in the widened expression. This paper presents such a solution, which is based on the following contributions:

- To express the set of operations that can be implemented on a particular target machine, we use *bitwidth types* borrowed from λ -RTL (Ramsey and Davidson, 1998).
- To express mathematical truths about how different operations use and produce high bits, we present *fill types*. An operator has a fill type only if it is *widenable*, i.e., if applying the operator to wide values can simulate the same operator applied to narrow values. In this case, the operator's fill type formalizes what it requires in the high bits of its arguments and what it guarantees about the high bits of its result.
- To widen an expression while minimizing the number of explicit extensions, we present a translation expressed as a set of inference rules. The rules specify a translation of an input expression, which operates on narrow values, into an output expression, which operates on values that are supported by the target machine. Each rule is assigned a cost; a translation with minimal cost is computed by dynamic programming.

2 Machine-Level Computation at Multiple Widths

Our semantic model, which is based on the machine-description language λ -RTL (Ramsey and Davidson, 1998), is that a machine operates on bit vectors. It takes bit vectors from memory, registers, or immediate fields; it applies low-level operators to compute new bit vectors; and finally it stores the results in registers or memory. Bit vectors are untyped; type distinctions, such as the distinction between signed and unsigned integers, are entirely in the operators.

In λ -RTL, most operators are polymorphic in width. For example, xor accepts two n -bit values and produces an n -bit result, for any n . Polymorphism is essential for a metalanguage that must describe machines not yet built, but for widening, we need to reason only about particular instances of polymorphic operators: those that appear in the source code or that can be used on a particular target machine. Even one target machine may support multiple instances of an operator; for example, the Pentium provides three instances of integer addition, which operate on 8-bit, 16-bit, and 32-bit values.

As in λ -RTL, the type of an operator is determined by the width (number of bits) of its arguments and result. These widths can be used to classify operators.

- A typical operator takes two n -bit arguments and produces a result of the same width, or possibly a Boolean result, which in this paper we represent as a result of width 1.¹ We call such an operator an *ordinary value operator*.
- A less typical operator might produce take n -bit arguments and produce a result of a different width; for example, multiplying two n -bit arguments may produce a $2n$ -bit result. Such an operator may change widths, but its primary purpose is to compute values, so we call it an *unusual value operator*.
- Three operators—sign extension, zero extension, and truncation—exist only to change widths. We call them *width-changing operators*.

The width-changing operators play a central role in widening. To make a narrow value wider, we can *sign extend* or *zero extend*.

sign extend $\text{sx}_{w \leftarrow n} v$ fills the $(w - n)$ high bits of the result with copies of v 's high (sign) bit. The subscript $w \leftarrow n$ is pronounced “ w from n .”

$$v' = \text{sx}_{8 \leftarrow 4} v \quad \begin{array}{|c|c|c|c|} \hline v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v$$

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline v_3 & v_3 & v_3 & v_3 & v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v'$$

zero extend $\text{zx}_{w \leftarrow n} v$ fills the $(w - n)$ high bits of the result with zeroes.

$$v' = \text{zx}_{8 \leftarrow 4} v \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v$$

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v'$$

Each extension preserves the n least significant bits of its operand.

To make a wide value narrower, we can only *truncate*.

truncate $\text{lo}_{n \leftarrow w} v$ extracts the n low bits of v . The high bits are lost.

$$v' = \text{lo}_{4 \leftarrow 8} v \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline v_7 & v_6 & v_5 & v_4 & v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v$$

$$\begin{array}{|c|c|c|c|} \hline v_3 & v_2 & v_1 & v_0 \\ \hline \end{array} = v'$$

Value operators at multiple widths. If we are given a program that uses an operator at a narrow width n , but the target machine provides that operator only at a wider width w , what can we do? In almost all cases we can use the wide version of the operator. For example, 32-bit add can be implemented by 64-bit add, and 32-bit xor can be implemented by 64-bit xor, regardless of what is in the high bits of the operands. As another example, 32-bit unsigned divide can be implemented by 64-bit unsigned divide, but only if the high bits of the operands are zero. In general, different operators put different constraints on the high bits of their operands, so we have to know what is in those high bits. When

¹ Full λ -RTL includes a Boolean type, but in this paper we simplify by representing a Boolean value as a bit vector of width 1.

operators are nested, results become operands, so we also have to know about high bits of results. For example, if xor is given operands with zeroes in the high bits, it produces a result with zeroes in the high bits, and similarly for unsigned divide. But two's-complement addition produces a value whose high bits can't be relied on: In a typical case, the high bits may be sign bits, but if the add overflows, the high bits are garbage.

These properties of operators, operands, and results are truths of mathematics; they are independent of any program or machine. We express these truths using a type system. The type of a value tells us whether its high bits are filled with sign bits, zeroes, or garbage, so call the types *fill types*. The fill type of an operator tells us what it needs in the high bits of its operands and what it provides in the high bits of its result.

In translating an expression, we can force an operand to have the right fill type by inserting an explicit sign or zero extension. But if the operand already has the right fill type, we can do without an explicit extension. For example, the following two expressions produce the same result.

$$\begin{aligned} &\text{divu}(\text{zx}_{64 \leftarrow 16}(\text{lo}_{16 \leftarrow 64}(\text{divu}(\text{zx}_{64 \leftarrow 16} x_{16}, \text{zx}_{64 \leftarrow 16} y_{16}))), k_{64}) \\ &\text{divu}(\quad \quad \quad \text{divu}(\text{zx}_{64 \leftarrow 16} x_{16}, \text{zx}_{64 \leftarrow 16} y_{16}) , k_{64}) \end{aligned}$$

The second expression avoids a superfluous zero extension.

Sometimes we can choose where to put an extension. Both xor and divu preserve zeroes in the high bits, so these expressions produce the same result.

$$\begin{aligned} &\text{divu}(\text{zx}_{64 \leftarrow 32}(\text{xor}(x_{32}, \quad \quad \quad y_{32})), \quad \quad \quad k_{64}) \\ &\text{divu}(\quad \quad \quad \text{xor}(\text{zx}_{64 \leftarrow 32} x_{32}, \text{zx}_{64 \leftarrow 32} y_{32}), k_{64}) \end{aligned}$$

Finally, many operators can be used at more than one fill type. If we choose fill types cleverly, we can avoid some extensions entirely. For example, in the somewhat fanciful expression

$$\text{popcnt}(\text{and}(\text{neg}(x_{32}), \text{divu}(y_{32}, k_{32}))),$$

popcnt counts the number of one bits in the value of a complex expression. To zero-extend each intermediate result would be valid but naïve. The constant k can be extended statically, so we need to extend at most variables x and y .

These examples show that different choices lead to different numbers of extensions in widened code. The number of extensions is the *cost* of widening, and we want to minimize it. Our implementation minimizes costs by dynamic programming (Sec. 7).

Which operators can be widened? Building on earlier work by Ramsey and Davidson (1998), we have identified 72 machine-level operators. These operators suffice to describe the instruction sets of a variety of machines, so chances are they include the operators one would find in any low-level intermediate code. Almost all operators are *widenable*, which means that a wide instance can do the job of a narrow instance (a formal definition appears in Sec. 3). There are 8 exceptions.

Expression $e, b ::= x_n$	Metavariable	Stands for	Domain
k_n	e, b	expression	syntax
$\text{sx}_{w \leftarrow n} e$	k	bit vector	syntax
$\text{zx}_{w \leftarrow n} e$	v	bit vector	semantics
$\text{lo}_{n \leftarrow w} e$	n, w	natural number	semantics
$\text{sxlo}_{w \leftarrow n}(b, e)$	\oplus	binary operator	both
$\text{zxlo}_{w \leftarrow n}(b, e)$	Metavariables n (narrow) and w (wide) give the widths of bit vectors. By convention, $n \leq n' \leq w \leq w'$.		
Assignment $a ::= x_w :=_w e$			

Fig. 2. Abstract syntax and metavariable conventions of WL

- Bit-rotate left and bit-rotate right cannot be widened. A bit-rotate operator must be translated into a combination of shift left, shift right, and bitwise or operators, each of which can be widened individually.
- The 6 operators that detect overflow of integer addition, subtraction, multiplication, and division cannot be widened. Instead, they must be translated to detect “narrow overflow.” Sec. 8 gives a sample translation.
- All 31 of the remaining integer-arithmetic, comparison, and logical operators can be widened, including the carry and borrow operators needed to implement multi-precision arithmetic.
- Of the 72 operators, 33 involve floating-point numbers. Floating-point operators can be widened (e.g., from 32 bits to 64 bits or from 64 bits to 80 bits), but the results produced by widened code only approximate the results from the original code, because intermediate results have more precision than originally specified. The details are beyond the scope of this paper.

Widening a program. In practice, we widen not just a single operation but an entire program. A widened program uses only operators instantiated at widths that are actually supported by the hardware, but it has almost the same effect as the original program: In particular, the low bits of all variables are the same as in the original program. If the high bits of some variables are garbage, the widened program may leave different garbage in those high bits.

3 WL: A Language to Model Widening

Because widening is a local transformation, widening a program is equivalent to widening every assignment in that program. We therefore model widening using a small language named WL, which has only assignments. An assignment has the form $x := e$, where the width of x is fixed by the program text, and operators in expression e may be nested arbitrarily deeply. Our translation handles assignments in which integer, logical, and comparison operators are used at any width up to the width of the target machine.

Fig. 2 shows the syntax of WL. An expression is a variable, bit-vector literal, sign or zero extension, truncation, or binary-arithmetic operation.

- A variable x_n stands for a mutable location that is n bits wide. On a typical machine, such a location is in a register or in memory.
- A literal k_n is a constant vector of n bits. We use decimal notation, but it stands for a two’s-complement representation.
- A bit vector may be interpreted as a natural number (unsigned integer); we use the denotation function \mathcal{U}_n , where $\mathcal{U}_n(v) = \sum_{i=0}^{n-1} 2^i v_i$.
- Operation $\text{sx}_{w \leftarrow n} e$ sign extends the n -bit expression e to w bits. Operation $\text{zx}_{w \leftarrow n} e$ similarly zero extends e . Operation $\text{lo}_{n \leftarrow w} e$ extracts the least significant n bits of the w -bit expression e . All three operations are explained in Sec. 2 above, with diagrams. On a typical machine, operations sx and zx are included in special-purpose load instructions such as “load signed byte” or “load halfword.”

In these width-changing operations, the left arrow is pronounced “from;” the right-to-left notation simplifies our proofs by making it easier to recognize and apply axioms such as $\text{sx}_{w' \leftarrow w}(\text{sx}_{w \leftarrow n} e) = \text{sx}_{w' \leftarrow n} e$.

- Operations $\text{sxlo}(b, e)$ and $\text{zxlo}(b, e)$ extend the least significant b bits of expression e ; to take an example, $\text{sxlo}_{32 \leftarrow 32}(3_{32}, 7_{32}) = -1_{32}$. These operations can be thought of as composing extension with truncation, but they are not equivalent because b , the number of bits to be extended, may be computed dynamically. On a typical machine, the sxlo and zxlo operations do not change the widths of their operands, i.e., the only instances available are $\text{sxlo}_{w \leftarrow w}$ and $\text{zxlo}_{w \leftarrow w}$. Usually sxlo is implemented as left shift followed by arithmetic right shift, and zxlo is implemented as left shift followed by logical right shift.

When the properties of the target machine are taken into account, sxlo may be more expressive than a combination of sx and lo. For example, the SPARC provides $\text{sx}_{32 \leftarrow 8}$, $\text{sx}_{32 \leftarrow 16}$, $\text{lo}_{8 \leftarrow 32}$ and $\text{lo}_{16 \leftarrow 32}$, and with these operators alone it is not possible to sign extend the least significant 13 bits of a 32-bit register. But the SPARC also provides $\text{sxlo}_{32 \leftarrow 32}$, so we can perform this extension using $\text{sxlo}_{32 \leftarrow 32}(13_{32}, e)$.

- To state properties that apply equally to both sx and zx or to sxlo and zxlo, we use the shorthands σx and σxlo .
- Operation $\oplus(e_1, e_2)$ applies the binary operator \oplus to expressions e_1 and e_2 . Operator \oplus may stand for add, subtract, multiply, divide, xor, bitwise and, bitwise or, or other operators—a complete list appears in Fig. 4.

We limit ourselves to binary operators only to keep notation simple. Our results apply to operators of any arity, and Fig. 4 presents our findings on all operators, not only binary ones.

WL’s operational semantics appears in a companion technical report (Redwine and Ramsey, 2004).

4 Bitwidth Types

Every expression in WL evaluates to a bit vector. On a particular target machine, bit vectors are restricted to certain widths. The width of the vector computed by an expression is given by that expression's *bitwidth type*. Our system of bitwidth types, which is a simplification of λ -RTL's type system (Ramsey and Davidson, 1998), involves two formal judgments. Judgment $M \vdash e : w$ says that in machine environment M , expression e is well typed with width w . Judgment $M \vdash x_w :=_w e$ says that assignment $x_w :=_w e$ is well typed in machine environment M . The rules are straightforward and are relegated to the companion technical report. The interesting part is the machine environment M .

The machine environment tells us at what widths each operator is available on a particular machine. Because an operator may be available at more than one type, we represent a machine environment M as a set of bindings of the form $\oplus : \text{width} \times \text{width} \rightarrow \text{width}$. For example, today's typical desktop machine might include the bindings $\text{add} : 32 \times 32 \rightarrow 32$ and $\text{lt} : 32 \times 32 \rightarrow 1$.

The machine environment M contains bindings only for instances available on a particular machine. We also define M^∞ , the *complete* machine environment, which contains all possible instances of each polymorphic λ -RTL operator. M^∞ enables us to type-check an expression independent of the target machine; it is the price we pay for eliminating polymorphism from our formal system. For all target machine environments M , $M \subset M^\infty$.

If the bindings in a machine environment tell the truth about the operators, we call that environment *sound*.

Definition 1. M is sound iff whenever $\oplus : w_1 \times w_2 \rightarrow w \in M$, v_1 is w_1 bits wide, v_2 is w_2 bits wide, and $\oplus_{w \leftarrow w_1 \times w_2}(v_1, v_2) = v$, then value v is w bits wide.

An environment is *well formed* if for each operator, the widths of operands determine the width of the result. In a well-formed environment every expression has at most one bitwidth type.

Definition 2. M is well formed when, for any operator \oplus ,

$$\oplus : w_1 \times w_2 \rightarrow w_3 \in M \wedge \oplus : w_1 \times w_2 \rightarrow w_4 \in M \text{ only if } w_3 = w_4 .$$

All environments in this paper, including M^∞ , are sound and well formed.

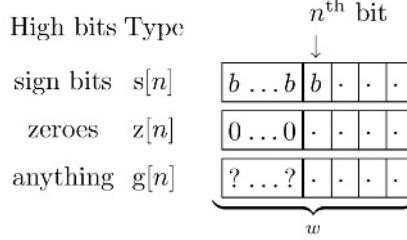
5 Fill Types for High Bits

A fill type of an expression tells us about the high bits of the bit vector it evaluates to. If we know that the result of evaluating an expression will have sign bits, zeroes, or garbage in the high bits, we can give that expression a fill type of s, z, or g. Formally, to say that all but the least significant n bits of e are copies of the n th (sign) bit, we write $e :: s[n]$. If all but the least significant n bits of e are zero, we write $e :: z[n]$. If we know nothing about the high bits of e 's value,

FILL-VAR ($n \leq w$)	$\frac{\Gamma(x_w) = \tau[n]}{\Gamma \vdash x_w :: \tau[n]}$	FILL-EXTLO ($n' \leq w$)	$\frac{}{\Gamma \vdash \sigma_{xlo_{w \leftarrow n'}}(b, e) :: \sigma[n']}$
FILL-LIT ($n \leq w$, $\text{fits}_{\tau[n], w} k_w$)	$\frac{}{\Gamma \vdash k_w :: \tau[n]}$	FILL-EXTLO-LIT $(n \leq n' \leq w, n = \mathcal{U}_{n'}(k_{n'}))$	$\frac{}{\Gamma \vdash \sigma_{xlo_{w \leftarrow n'}}(k_{n'}, e) :: \sigma[n]}$
FILL-EXT-PRESERVE ($n \leq n' \leq w$)	$\frac{\Gamma \vdash e :: \sigma[n]}{\Gamma \vdash \sigma_{x_{w \leftarrow n'}} e :: \sigma[n]}$	FILL-BINOP	
		$\frac{\Gamma \vdash e_1 :: \tau_1[n_1] \quad \Gamma \vdash e_2 :: \tau_2[n_2]}{\oplus :: \tau_1 \times \tau_2 \rightarrow \tau}$	
		$\frac{\oplus : n_1 \times n_2 \rightarrow n \in M^\infty}{\Gamma \vdash \oplus_{m \leftarrow m_1 \times m_2}(e_1, e_2) :: \tau[n]}$	
FILL-EXT ($n' \leq w$)	$\frac{}{\Gamma \vdash \sigma_{x_{w \leftarrow n'}} e :: \sigma[n']}$	FILL-GARBAGE	
FILL-LO-PRESERVE ($n \leq n' \leq w$)	$\frac{\Gamma \vdash e :: \tau[n]}{\Gamma \vdash \text{lo}_{n' \leftarrow w} e :: \tau[n]}$	$\frac{}{\Gamma \vdash e :: g[n]}$	
FILL-EXTLO-PRESERVE ($n \leq n' \leq w$)	$\frac{\Gamma \vdash e :: \sigma[n]}{\Gamma \vdash \sigma_{xlo_{w \leftarrow n'}}(b, e) :: \sigma[n]}$	FILL-SUBSUME-INDEX ($n \leq n'$)	
		$\frac{\Gamma \vdash e :: \sigma[n]}{\Gamma \vdash e :: \sigma[n']}$	

Fig. 3. Fill typing rules

we can write $e :: g[n]$ for any e . These types can be shown pictorially as follows:



Each fill type corresponds to a semantic predicate. Given a bit vector v of width w , $\text{fits}_{s[n], w} v$ is true iff the high $w - n$ bits of v are copies of the sign bit. Similarly, $\text{fits}_{z[n], w} v$ is true iff the high $w - n$ bits of v are zero. Formally, $\text{fits}_{s[n], w} v$ iff $v = s_{w \leftarrow n}(\text{lo}_{n \leftarrow w} v)$ and $\text{fits}_{z[n], w} v$ iff $v = z_{w \leftarrow n}(\text{lo}_{n \leftarrow w} v)$. Finally, $\text{fits}_{g[n], w} v$ is always true.

To abbreviate any fill type, we write τ instead of s , z , or g . To abbreviate only a non-garbage fill type, we write σ instead of s or z . The formal rules for fill types appear in Fig. 3. The formal judgment of the system is $\Gamma \vdash e :: \tau[n]$, where Γ is an environment that gives the fill type of each variable.

- A variable has the fill type assigned to it by Γ . We explain where Γ comes from in Sec. 6.1.

- A constant may have any fill type consistent with its value.
- An extension may preserve an existing fill type or establish a new fill type.
- Truncation preserves fill type, provided not too many bits are truncated.
- Just like $\sigma x_{w \leftarrow n'}$, $\sigma x lo_{w \leftarrow n'}$ may preserve an existing fill type or establish a new one. If the first argument is a literal constant, $\sigma x lo_{w \leftarrow n'}$ may also establish a stronger fill type.
- The rule for binary operators is the most complex and also the most important rule; it is discussed below.
- The rules below the black bar express subtyping relations on fill types. The subtyping relations are a consequence of the fits predicate defined above. Garbage subsumes other types since it is always true. The meaning of the extension operations implies that if $\text{fits}_{\sigma[n], w}$ then $\text{fits}_{\sigma[n+1], w}$ for $n < w$, which explains the index subtypes.

Fill types of operators. If a binary operator has a fill type, it takes the form $\tau_1 \times \tau_2 \rightarrow \tau$. The fill type is meaningful in conjunction with the operator’s bitwidth type: If the operator is instantiated at bitwidth type $n_1 \times n_2 \rightarrow n$, then provided the arguments have fill types $\tau_1[n_1]$ and $\tau_2[n_2]$, the result has fill type $\tau[n]$. We give an operator a fill type only if that operator is *widenable*. Widenability promises not only that the result has the right fill type, but also that the least significant n bits of the result are the bits that would have been produced by applying the narrow instance of the operator to the least significant bits of the operands.

Definition 3. An operator \oplus is widenable at fill type $\tau_1 \times \tau_2 \rightarrow \tau$, written $\oplus :: \tau_1 \times \tau_2 \rightarrow \tau$, if whenever M^∞ contains the two bindings $\oplus : n_1 \times n_2 \rightarrow n$ and $\oplus : w_1 \times w_2 \rightarrow w$, where $n_1 \leq w_1$, $n_2 \leq w_2$, and $n \leq w$, then for all v_1, v_2 such that $\text{fits}_{\tau_1[n_1], w_1} v_1$ and $\text{fits}_{\tau_2[n_2], w_2} v_2$, both $\text{fits}_{\tau[n], w}(\oplus_{w \leftarrow w_1 \times w_2}(v_1, v_2))$ and

$$\text{lo}_{n \leftarrow w}(\oplus_{w \leftarrow w_1 \times w_2}(v_1, v_2)) = \oplus_{n \leftarrow n_1 \times n_2}(\text{lo}_{n_1 \leftarrow w_1} v_1, \text{lo}_{n_2 \leftarrow w_2} v_2).$$

The second condition should also be taken to mean that $\oplus_{w \leftarrow w_1 \times w_2}(v_1, v_2)$ is well defined whenever $\oplus_{n \leftarrow n_1 \times n_2}(\text{lo}_{n_1 \leftarrow w_1} v_1, \text{lo}_{n_2 \leftarrow w_2} v_2)$ is: If a narrow computation completes successfully, so does a wide one.

Widenability is needed to show that the rules in Fig. 3 correctly predict the high bits of results—a property we call fill-type soundness. Soundness also requires a dynamic environment that is consistent with the fill-type environment: each variable must have the fill type claimed. Formally, ρ is *consistent* with Γ iff $\forall x_w \in \text{dom } \Gamma, x_w \in \text{dom } \rho \wedge \text{fits}_{\Gamma(x_w), w}(\rho(x_w))$.

Theorem 4. (Fill-Type Soundness) If $\Gamma \vdash e :: \tau[n]$, $\rho \vdash e \Downarrow v$, $M^\infty \vdash e : w$, and ρ is consistent with Γ , then $\text{fits}_{\tau[n], w} v$.

Widenability is a key idea: If an operator is widenable at a suitable fill type, a wide instance of the operator can do the job of a narrow instance. The examples in Sec. 2 are possible only because operators `popcnt`, `neg`, `divu`, `bitwise and`, and `xor` are widenable. The fill types of a widenable operator express all

add ::	$g \times g \rightarrow g$	mod ::	$s \times s \rightarrow s$
add_overflows Not widenable		modu ::	$z \times z \rightarrow z$
and ::	$s \times s \rightarrow s$	mul ::	$s \times s \rightarrow s$
and ::	$z \times g \rightarrow z$	mulu ::	$z \times z \rightarrow z$
and ::	$g \times z \rightarrow z$	mul_overflows Not widenable	
and ::	$g \times g \rightarrow g$	mulu_overflows Not widenable	
borrow ::	$s \times s \times g \rightarrow z$	ne ::	$s \times s \rightarrow z$
borrow ::	$z \times z \times g \rightarrow z$	ne ::	$z \times z \rightarrow z$
carry ::	$s \times s \times g \rightarrow z$	neg ::	$g \rightarrow g$
com ::	$s \rightarrow s$	or ::	$s \times s \rightarrow s$
com ::	$g \rightarrow g$	or ::	$z \times z \rightarrow z$
div ::	$s \times s \rightarrow s$	or ::	$g \times g \rightarrow g$
div_overflows Not widenable		popcnt ::	$z \rightarrow z$
divu ::	$z \times z \rightarrow z$	quot ::	$s \times s \rightarrow s$
eq ::	$s \times s \rightarrow z$	quot_overflows Not widenable	
eq ::	$z \times z \rightarrow z$	rem ::	$s \times s \rightarrow s$
ge ::	$s \times s \rightarrow z$	rotl ::	Not widenable
geu ::	$s \times s \rightarrow z$	rotr ::	Not widenable
geu ::	$z \times z \rightarrow z$	shl ::	$g \times z \rightarrow g$
gt ::	$s \times s \rightarrow z$	shra ::	$s \times z \rightarrow s$
gtu ::	$s \times s \rightarrow z$	shrl ::	$z \times z \rightarrow z$
gtu ::	$z \times z \rightarrow z$	sub ::	$g \times g \rightarrow g$
le ::	$s \times s \rightarrow z$	sub_overflows Not widenable	
leu ::	$s \times s \rightarrow z$	xor ::	$s \times s \rightarrow s$
leu ::	$z \times z \rightarrow z$	xor ::	$z \times z \rightarrow z$
lt ::	$s \times s \rightarrow z$	xor ::	$g \times g \rightarrow g$
ltu ::	$s \times s \rightarrow z$		
ltu ::	$z \times z \rightarrow z$		

Fig. 4. Widenability and fill types of integer and logical operators

the mathematical facts we need to know to widen expressions involving that operator. Fig. 4 lists the fill types and widenability of all non-floating-point operators. The fill type used for the results of comparisons is arbitrary, because a 1-bit value is always at its natural width. We have chosen the z type.

A few operators can illustrate the interpretation of fill types. Unsigned division (divu) has a fill type typical of unsigned operators. It requires both arguments to have zeroes in the high bits and guarantees a result with zeroes in the high bits. Addition (add) requires nothing of the high bits of its operands but guarantees nothing about the high bits of its result. Exclusive or (xor) is overloaded. It can accept any input, but guarantees to produce zeroes in the high bits of its result only if the high bits of each operand are zero and sign bits only if each operand has sign bits. Unsigned greater-than (gtu), perhaps surprisingly, is also overloaded: it can be used to compare sign-extended numbers as well as zero-extended numbers. It is an accident of two's-complement representation that extending the sign bit does not change the results of this comparison.

6 Translating a Narrow Program for a Wide Machine

To translate a narrow program to run on a wide machine takes two steps:

Variable placement: Decide in what location on the target machine each source variable will be stored.

Widening: Rewrite the program to use wide operators in place of narrow operators.

Although variable placement and widening can be combined, it is simpler to place variables before widening, as we have done in our implementation.

6.1 Placing Variables

A variable placer chooses a location for each variable in the source program. Given a variable x_n of width n , there are two choices: place it in a target location of width n , which we also call x_n , or place it in the low n bits of a target location of width w , which we call x_w . (A location of width w may be used only if the variable stands for a location in a machine register; a variable placer should not change the layout of data structures in memory.) If we choose the low n bits of x_w , we have to decide what to do with the high bits. We express this decision by giving x_w a fill type, which we store in type environment Γ . For example, a Java signed 32-bit integer variable x might be placed in a 64-bit location, in which case $\Gamma(x_{64}) = s[32]$.

The best choice of $\Gamma(x_w)$ depends on how x_w is used in the program. For example, if x_w is frequently passed to operators that require sign bits in the high bits, it may be best if $\Gamma(x_w) = s[n]$. If x_w is frequently passed to operators that place no requirements on high bits, $\Gamma(x_w) = g[n]$ may be best. Finding the best placement requires global knowledge about how each variable is used in each assignment and how frequently each assignment is executed.

Another interesting way to choose $\Gamma(x_w)$ is to use bitwidth inference (Razdan and Smith, 1994; Stephenson et al., 2000; Budiu et al., 2000). Bitwidth inference finds the range of values each variable can have. If the range is nonnegative, we use the z type; otherwise we use the s type. In either case,

the range is used to determine the minimum bitwidth necessary to represent the variable. This bitwidth becomes the index of the variable's fill type $\Gamma(x_w)$.

Kawahito et al. (2002) describe a clever variation on placement that can be used to improve performance. At a definition of x_w , they assume $\Gamma(x_w) = s[n]$, but at a use, they assume $\Gamma(x_w) = g[n]$. These assumptions can yield a program with extension operations at both definitions and uses, which would be redundant. The advantage is that the redundancy can be removed from *either* the uses or the definition, whichever is more frequently executed. Our method can support this technique, but to keep things clear we suppress the details.

6.2 Widening

After a location has been chosen for each variable, the *widener* replaces each narrow operation with a wide operation or combination of operators. An operator that is widenable can simply be used at the new width, provided its operands are given suitable fill types. An operator that is not widenable must be rewritten as a combination of widenable operators. As discussed in Sec. 2, there are only a handful of non-widenable operators, and each can easily be rewritten.

Widening is defined by a formal system that yields judgments of the form

$$\Gamma, M \vdash e \xrightarrow{c} e' :: \tau[n] : w.$$

This judgment says that given variable placement Γ and target machine M , expression e can be widened to expression e' at cost c , and that e' has fill type $\tau[n]$ and width w . A similar judgment, $\Gamma, M \vdash a \xrightarrow{c} a'$, applies to assignments.

Given Γ , M , e , τ , n , and w , the rules in Fig. 5 find every e' that can be found by inserting sign or zero extensions. Our implementation uses dynamic programming to find an e' of minimum cost.

The rules in Fig. 5 are divided into four groups.

- The upper-left rules translate those parts of the abstract syntax that relate to value computation without widening or narrowing. The assignment rule, in particular, shows that the left-hand side's width w and fill type $\tau[n]$ provide the information needed to translate the right-hand side. The LIT rule shows that we can extend a literal constant at translation time. The BINOP rule is the primary source of nondeterminism in the system. Nondeterminism arises for those operators that have multiple fill types, such as exclusive or. Nondeterminism may also arise for an operator that the target machine supports at multiple widths. For example, if the source code contains a 12-bit add and the target machine is the Pentium, the widener can do the add at either 16 or 32 bits.

The BINOP rule shows where our two type systems come together. The fill types of an operator are universal truths of mathematics, but the bitwidth types of that operator are accidental facts about the machine.

The BINOP rule combines both kinds of information.

$\text{ASSIGN } (n \leq w)$ $\frac{\Gamma(x_w) = \tau[n]}{\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: \tau[n] : w}{\Gamma, M \vdash x_w :=_w e \xrightarrow{c} x_w :=_w e'}}$	$\text{DROP-EXT } (n \leq n' \leq w)$ $\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: \sigma[n] : w}{\Gamma, M \vdash \sigma x_{n' \leftarrow n} e \xrightarrow{c} e' :: \sigma[n] : w}$
$\text{VAR } (n \leq w)$ $\frac{\Gamma(x_w) = \tau[n]}{\Gamma, M \vdash x_w \xrightarrow{0} x_w :: \tau[n] : w}$	$\text{DROP-LO-COPY } (n \leq n' \leq w)$ $\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: \tau[n'] : w}{\Gamma, M \vdash \text{lo}_{n \leftarrow n'} e \xrightarrow{c} e' :: \tau[n'] : w}$
$\text{LIT } (k'_w = \sigma x_{w \leftarrow n} k_n, n \leq w)$ $\frac{\Gamma, M \vdash k_n \xrightarrow{0} k'_w :: \sigma[n] : w}{\Gamma, M \vdash k'_w :: \sigma[n] : w}$	
$\text{BINOP } (n_1 \leq w_1, n_2 \leq w_2, n \leq w)$ $\frac{\begin{array}{l} \oplus :: \tau_1 \times \tau_2 \rightarrow \tau \\ \oplus : n_1 \times n_2 \rightarrow n \in M^\infty \\ \oplus : w_1 \times w_2 \rightarrow w \in M \\ \Gamma, M \vdash e_1 \xrightarrow{c_1} e'_1 :: \tau_1[n_1] : w_1 \\ \Gamma, M \vdash e_2 \xrightarrow{c_2} e'_2 :: \tau_2[n_2] : w_2 \end{array}}{\frac{\Gamma, M \vdash \oplus_{n \leftarrow n_1 \times n_2}(e_1, e_2) \xrightarrow{c_1 \oplus c_2} \oplus_{w \leftarrow w_1 \times w_2}(e'_1, e'_2) :: \tau[n] : w}$	$\text{DROP-LO-IGNORE } (n \leq n' \leq w)$ $\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: \tau[n'] : w}{\Gamma, M \vdash \text{lo}_{n \leftarrow n'} e \xrightarrow{c} e' :: g[n] : w}$
$\text{FILL } (\mathcal{U}(k_w) = n, n \leq w \leq w')$ $\frac{\begin{array}{l} \Gamma, M \vdash e \xrightarrow{c} e' :: g[n] : w \\ \sigma x_{w' \leftarrow w} \in M \end{array}}{\Gamma, M \vdash e \xrightarrow{c+1} \sigma x_{w' \leftarrow w}(k_w, e') :: \sigma[n] : w'}$	$\text{EXTLO } (n \leq n' \leq w \leq w')$ $\frac{\begin{array}{l} \sigma x_{w' \leftarrow w} \in M \\ \Gamma, M \vdash b \xrightarrow{c_1} b' :: z[n] : w \\ \Gamma, M \vdash e \xrightarrow{c_2} e' :: g[n] : w \end{array}}{\frac{\Gamma, M \vdash \sigma x_{n' \leftarrow n}(b, e) \xrightarrow{c_1 \oplus c_2 + 1} \sigma x_{w' \leftarrow w}(b', e') :: \sigma[n] : w'}{\Gamma, M \vdash \sigma x_{w' \leftarrow w}(b', e') :: \sigma[n] : w'}}$
$\text{WIDEN-}\sigma \ (n \leq w \leq w')$ $\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: \sigma[n] : w}{\Gamma, M \vdash e \xrightarrow{c+1} \sigma x_{w' \leftarrow w} e' :: \sigma[n] : w'}$	$\text{SUBSUME-FILL } (n \leq w)$ $\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: \sigma[n] : w}{\Gamma, M \vdash e \xrightarrow{c} e' :: g[n] : w}$
$\text{WIDEN-g } (n \leq w \leq w')$ $\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: g[n] : w}{\Gamma, M \vdash e \xrightarrow{c+1} \sigma x_{w' \leftarrow w} e' :: g[n] : w'}$	$\text{SUBSUME-INDEX } (n < n' \leq w)$ $\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: \sigma[n] : w}{\Gamma, M \vdash e \xrightarrow{c} e' :: \sigma[n'] : w}$
$\text{NARROW } (n \leq w \leq w')$ $\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: \tau[n] : w'}{\Gamma, M \vdash e \xrightarrow{c+1} \text{lo}_{w \leftarrow w'} e' :: \tau[n] : w'}$	NATURAL $\frac{\Gamma, M \vdash e \xrightarrow{c} e' :: g[w] : w}{\Gamma, M \vdash e \xrightarrow{c} e' :: \sigma[w] : w}$

Fig. 5. Widening rules

- The lower-left rules provide the means of changing an expression’s width. Each such rule requires an explicit extension or truncation. We need both FILL and $\text{WIDEN-}\sigma$ because, as noted in Sec. 3, we may have to use either σ_{xlo} or σ_{x} depending on what is available on the target machine. Each of these rules adds 1 to the cost of widening because it inserts an extension or truncation operation.
- The upper-right rules cope with extensions and truncations in the source program. If the number of bits extended or truncated is known statically, we simply drop the extension or truncation and let the other rules insert one if needed. In rule EXTLO , the number of bits extended is computed dynamically, so the extension must be retained. The DROP rules maintain the same cost because they insert nothing in the widened expression. EXTLO , on the other hand, has to add the extension from the source expression into the widened expression.
- The lower-right rules permit us to exploit the subtyping rules for fill types. They change the type without changing the widened expression, so they have no extra cost.

The rules are sound, which means that a translated expression e' produces a value with the fill type claimed and with the right answer in the low bits. Also, all the operators in any translated expression are available on the target machine. Stating these properties precisely requires a bit more formalism.

To formalize soundness of the rules for assignment, we must also account for potential differences in high bits when $\Gamma(x_w) = g[n]$.

Definition 5. Environments ρ and ρ' agree up to widths in Γ , written $\rho \sim_{\Gamma} \rho'$, when both ρ and ρ' are consistent with Γ and when for each variable x_w , $\rho(x_w)$ and $\rho'(x_w)$ are equal in their n low bits, where n is determined by $\Gamma(x_w)$. Formally, $\forall x_w \in \text{dom } \Gamma, \text{lo}_{n \leftarrow w}(\rho(x_w)) = \text{lo}_{n \leftarrow w}(\rho'(x_w))$, where $\Gamma(x_w) = \tau[n]$.

N.B. If $\Gamma(x_w) = s[n]$, then the high bits are equal to the n th bit, and because ρ and ρ' agree on the low n bits, $\rho(x_w)$ and $\rho'(x_w)$ must be identical. Similarly, if $\Gamma(x_w) = z[n]$, then the high bits are zero, and again $\rho(x_w)$ and $\rho'(x_w)$ must be identical. But if $\Gamma(x_w) = g[n]$, the high bits can differ before and after widening.

Soundness of translation says that a translated assignment produces almost the same results as the original, the only differences being those permitted to two environments that agree up to widths in Γ . The assignment soundness theorem is shown here pictorially as a commutative diagram.

$$\begin{array}{ccc} \rho_{\text{pre}} & \sim_{\Gamma} & \rho'_{\text{pre}} \\ x_w := e \Downarrow & & \Downarrow x_w := e' \\ \rho_{\text{post}} & \sim_{\Gamma} & \rho'_{\text{post}} \end{array}$$

To prove soundness, we have a lemma for expressions. Whenever the original expression has a value, the translated expression does too, and the value of the translated expression agrees with the original on the low bits and respects the semantic predicate required by its fill type.

Theorem 6. (Assignment Soundness) *If $\Gamma, M \vdash x_w :=_w e \leftrightarrow x_w :=_w e'$, and we are given any ρ_1 and ρ_2 such that $\rho_1 \sim_\Gamma \rho_2$, and if $\rho_1 \vdash x_w :=_w e \Downarrow \rho'_1$, then $\rho_2 \vdash x_w :=_w e' \Downarrow \rho'_2$ and $\rho'_1 \sim_\Gamma \rho'_2$.*

Lemma 7. (Expression Soundness) *For any sound and well-formed M and ρ consistent with Γ , if $\Gamma, M \vdash e \leftrightarrow e' :: \tau[n] : w$, $\rho \vdash e \Downarrow v$, and $M^\infty \vdash e : m$, then $\rho \vdash e' \Downarrow v'$, $\text{fits}_{\tau[n], w}(v')$, and $n \leq m$ implies $\text{lo}_{n \leftarrow w} v' = \text{lo}_{m \leftarrow m} v$.*

Proof is by induction on the structure of derivations.

Soundness relies on the claims made about widenability of operators (Fig. 4). We make these claims based on careful thought, confirmed by exhaustive machine checking on a subset of the operators at widths 8 and 16. A formal proof would be better, but such a proof would require a bit-level semantics for each operator, such as is provided by Bacon (2001), as well as substantial theorem-proving expertise. This sort of treatment is well beyond the scope of this paper or of a typical compiler; in our opinion, the game is not worth the candle.

The translation is not only sound but also has other useful properties: e' has the types claimed for it, and e' can be implemented on the target machine.

7 Implementation and Experimental Results

We have implemented widening by using dynamic programming to compute the minimum-cost translation for a given expression at a given width and fill type. Dynamic programming is certain to find a solution if one exists, always finds the best solution, and is easy to implement by following the structure of our rules. The critical implementation decisions were how to represent a solution and how to eliminate redundant solutions. These details are covered in the companion technical report (Redwine and Ramsey, 2004).

We compare the dynamic-programming implementation with a top-down, greedy algorithm. The greedy algorithm inserts sign extensions or zero extensions below each operator as required by that operator's fill type. If an operator has more than one fill type, the greedy algorithm picks one that requires fewer extensions immediately below the operator. The greedy algorithm may seem simpler than dynamic programming, but in practice, it was surprisingly difficult to write and to think about, and we are less confident in its correctness.

We added both algorithms to the Quick C-- compiler, a compiler for C-- (Peyton Jones et al., 1999; Ramsey and Peyton Jones, 2000). The implementation is about 600 lines of Objective Caml code, including both algorithms and some supporting infrastructure. We then widened some 32-bit programs from a small test suite distributed with the LCC compiler (Fraser and Hanson, 1995). For this preliminary experiment, we placed variables in memory with garbage in the high bits, so for any x , $\Gamma(x) = g[32]$. Quick C-- generates such poor code that measurements of run time are not meaningful, so as a proxy we evaluate effectiveness by counting the number of operations in the widened code.

Table 1. Dynamic programming versus a greedy algorithm on a subset of the LCC compiler test suite. Original column counts operations in WL produced by the compiler. Greedy and DP show the number of operations after widening (smaller is better). Reduction shows the percent reduction in operations, going from Greedy to DP.

File name	Original	Greedy	DP	Reduction
incr.c	17	51	30	24%
struct.c	20	70	36	49%
front.c	31	93	58	38%
8q.c	36	113	66	42%
sort.c	61	187	117	37%
wf1.c	65	201	124	38%
init.c	77	239	148	38%
cq.c	79	252	151	40%
stdarg.c	112	392	215	45%
yacc.c	118	350	209	40%
switch.c	120	369	235	36%
limits.c	213	681	422	38%

Table 1 shows the results. The “Original” column shows the number of operations in original code. The “Greedy” and “DP” columns show the total number of operations in widened code. The final column shows the percent reduction in the number of operations going from the greedy algorithm to the dynamic-programming algorithm. The reductions range from 24% to 49%, showing that the dynamic-programming algorithm produces much better results.

8 Discussion

We have identified several potential extensions to our system, but none of significant value. For example, it is possible to define a fill type $o[n]$, which means that a value has ones in all but the least significant n bits. While the Boolean operators preserve the o type in ways one might expect (dual to the z type), it is not useful because no operation requires an operand with ones in the high bits.

Another potential extension is to apply widening to floating-point computations. Sign extension and truncation have their analogs in instructions for floating-point conversions, but there is a big difference: If a floating-point computation is performed with extra precision and truncated only at the end, we get a different answer than if every intermediate result is truncated to use less precision. If exact answers are required, then, one cannot remove any explicit extensions or truncations, and our ideas offer no benefit relative to a naïve translation. If, however, the programmer wants approximate answers fast, one can perform entire computations at higher precision. Our C-- back end for the Pentium offers this option, widening 32-bit and 64-bit floating-point computations to the full 80 bits offered by the hardware.

Even without these extensions, it is surprising how much machinery is needed to solve a seemingly simple problem. We started out with much simpler ideas, among which the central one was to rewrite expressions by exploiting algebraic laws such as the following:

$$\begin{aligned}\text{lo}_{n \leftarrow w}(\text{sx}_{w \leftarrow n} x) &= x \\ \text{lo}_{n \leftarrow w}(\text{zx}_{w \leftarrow n} x) &= x \\ \text{sx}_{w \leftarrow n}(\text{mul}(x, y)) &= \text{mul}(\text{sx}_{w \leftarrow n} x, \text{sx}_{w \leftarrow n} y) \\ \text{zx}_{w \leftarrow n}(\text{mulu}(x, y)) &= \text{mulu}(\text{zx}_{w \leftarrow n} x, \text{zx}_{w \leftarrow n} y)\end{aligned}$$

Unfortunately, simple algebraic rewriting has two deficiencies:

- It is not always obvious what extension operator to pick when rewriting an expression: A choice made at an outer level may affect the cost of rewriting deeply nested subexpressions. Perhaps this problem could be addressed using dynamic programming, but then the simplicity of rewriting would be lost.
- There is no obvious way to deal with garbage in the high bits. Attempts to introduce a gx (“garbage extend”) operator lead to a warm feeling at first, but gx is a relation, not a function, and the math quickly becomes unwieldy.

Algebraic rewriting is still useful when dealing with the eight non-widenable operators, which perform bitwise rotations or overflow detection. Each application of a non-widenable operator must be translated, before widening, into an equivalent expression that uses only widenable operators. For example, the operator that determines if an addition overflows, add_overflows , can be rewritten using either of these equations:

$$\begin{aligned}\text{add_overflows}_n(x, y) &= \text{ne}(\text{sxlo}_{w \leftarrow w}(n, \text{sx}_{w \leftarrow n} x + \text{sx}_{w \leftarrow n} y), \text{sx}_{w \leftarrow n} x + \text{sx}_{w \leftarrow n} y) \\ \text{add_overflows}_n(x, y) &= \text{add_overflows}_w(\text{shl}(\text{sx}_{w \leftarrow n} x, w - n), \text{shl}(\text{sx}_{w \leftarrow n} y, w - n))\end{aligned}$$

where ne is inequality and shl is shift left.

Although the amount of machinery is surprising, our solution does provide a satisfying separation of concerns:

- The capabilities of the target machine are captured in the machine environment M .
- Global decisions about variables are captured in the fill-type environment Γ .
- Machine-independent truths about the widenability of operators are captured in their fill types (Fig. 4).

We hope that fill types, especially, will be of lasting value to anyone who has to think about running narrow programs on a wide machine.

Acknowledgements. This work has been funded in part by NSF grants CCR-0311482 and CCR-0096069. João Dias, Paul Govereau, Glenn Holloway, Reuben Olinsky, Sukyoung Ryu, and Chung-chieh Shan provided helpful comments on the paper.

References

David F. Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. In *Proceedings of the Joint ACM Java Grande/ISCOPE Conference*, pages 68–77, Stanford, California, June 2001.

Mihai Budiu, Majd Sakr, Kip Walker, and Seth C. Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the EuroPar 2000 European Conference on Parallel Computing*, volume 1900 of *LNCS*, pages 969–979, August 2000.

Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, Redwood City, CA, 1995. ISBN 0-8053-1670-1.

Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Effective sign extension elimination. *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 37(5):187–198, May 2002. ISSN 0362-1340.

Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 1–28. Springer Verlag, September 1999.

Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *LNCS*, pages 172–188. Springer Verlag, June 1998.

Norman Ramsey and Simon L. Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 35(5):285–298, May 2000.

Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, November 1994.

Kevin Redwine and Norman Ramsey. Widening integer arithmetic. Technical Report TR-01-04, Division of Engineering and Applied Sciences, Harvard University, January 2004.

Mark Stephenson, Jonathan Babb, and Saman P. Amarasinghe. Bitwidth analysis with application to silicon compilation. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 35(5):108–120, May 2000.

Stochastic Bit-Width Approximation Using Extreme Value Theory for Customizable Processors*

Emre Özer, Andy P. Nisbet, and David Gregg

Department of Computer Science

Trinity College, Dublin, Ireland

{emre.ozer, andy.nisbet, david.gregg}@cs.tcd.ie

Abstract. Application-specific logic can be generated with a balance and mix of functional units tailored to match an application's computational requirements. The area and power consumption of application-specific functional units, registers and memory blocks is heavily dependent on the bit-widths of operands used in computations. The actual bit-width required to store the values assigned to a variable during execution of a program will not in general match the built-in C data types with fixed sizes of 8, 16, 32 and 64 bits. Thus, precious area is wasted if the built-in data type sizes are used to declare the size of operands. A novel stochastic bit-width approximation technique is introduced to estimate the required bit-width of integer variables using **Extreme Value Theory**. Results are presented to demonstrate reductions in bit-widths, area and power consumption when the probability of overflow/underflow occurring is varied from 0.1 to infinitesimal levels. Our experimental results show that the stochastic bit-width approximation results in overall 32% reduction in area and overall 21% reduction in the design power consumption on a FPGA chip for nine embedded benchmarks.

1 Introduction

Unlike general-purpose processors, the bit-widths of variables, functional units and data buses in customizable processors such as FPGAs and ASICs can be tailored to meet the requirements of the application. The area requirements and power consumption can be significantly reduced through the use of such customized data bit-widths. Exploiting the potential of bit-width customization is a major problem when compiling high-level languages such as C directly to custom hardware. However, C has no facility for declaring the exact bit-widths of variables and it provides fixed data bit-widths such as byte, half-word and word, causing precious hardware resources to be occupied by unused bits.

One solution is to extend the C language with macro support so that the programmer can define the bit-widths of variables as done in [8 12 13 14]. However, this places a significant burden on the programmer to perform extensive analysis in order to specify the actual data bit-widths of each and every variable without a single occurrence of overflow or underflow during program execution. The approach advocated in

* The research described in this paper is supported by an Enterprise Ireland Research Innovation Fund Grant IF/2002/035.

this paper is a hardware compilation infrastructure from C to custom hardware that automatically calculates the data bit-widths of all integer variables including arrays, and both local and global variables. Hence, the programmers need not concern themselves with bit-width specifications, and they need only focus on the correct specification of the algorithm in C prior to passing their code to the compilation infrastructure.

The novel stochastic approximation technique calculates the bit-width of each variable in the application by estimating the probable value range using *Extreme Value Theory* [10 11] in Statistics. Not only can our technique estimate the bit-widths of the variables that are sensitive to input data, but also it is quite successful in determining the bit-widths of the variables inside loops (i.e. while loops) whose loop counts are unknown at compile time. However, the approximation of bit-widths is done with a finite probability for the occurrence of overflow or underflow. Hence, this technique is not intended to be applied to compiling general-purpose programs. Instead, it is best suited to embedded-type applications such as digital signal processing (DSP), speech, video, multimedia, and similar programs. Such code typically contains very regular control flow and easily-defined input sets. This allows us to generate large, meaningful data sets without worrying about coverage or termination when performing sample runs. The results of these runs are fed into our compiler, which uses statistical analysis techniques to estimate the bit-widths of variables.

It is also important to note that we do not determine the necessary bit-width to guarantee that there is no overflow or underflow in variables. Using our technique, there is always a finite probability that a variable may encounter over/underflow. The strength of our technique lies in using statistical techniques to estimate the sufficient bit-width by reducing the probability of over/underflow to infinitesimally small values. Although the probability of over/underflow always remains, it can be reduced to such a tiny level that the probability of device failure or memory errors is significantly higher than the probability of over/underflow in a variable.

The organization of the paper is as follows. *Section 2* introduces stochastic bit-width approximation. *Section 3* presents our experimental framework and nine embedded benchmarks. *Section 4* explains the results from the experimental statistical analysis and *Section 5* analyzes the results of area and power consumption acquired from actual FPGA implementation of the benchmarks. *Section 6* shows the related work, and *Section 7* concludes the paper.

2 Stochastic Bit-Width Approximation

The value of a variable can be constant during the execution of the entire program or can depend on the inputs or other variables that may in turn depend on the inputs. The optimal data range between the maximum and minimum values assigned to a variable must be determined in order to find its optimal bit-width. The *optimal bit-width* of a variable is defined as the precision where no overflow or underflow can occur when values are assigned to a variable by running the program with all possible data combinations of its inputs. The optimal bit-width of a variable holding a constant value, or a finite set of different constants can be determined by the maximum optimal bit-width of the set of constants, regardless of data inputs. However, finding the optimal bit-width of a variable that depends on data inputs is an exhaustive search problem with exponential time complexity. For example, consider a program with one input

variable whose value covers the range $[-2^{n-1}, 2^{n-1} - 1]$ in two's complement representation. The number of all possible input combinations is $M = 2^s$ where s represents the number of input elements and n is the bit-width of the variable. The application under observation must execute each one of M data inputs and observe the extreme values of each variable. This is clearly an exhaustive search with exponential time complexity. Given such complexity, we have chosen to use stochastic approximation to estimate *the probable value range* of a variable. *The probability of over/underflow* is defined as the probability of a value being assigned to a variable that is outside of its probable value range. The stochastic bit-width approximation estimates the probable value range of all variables with a finite probability of overflow/underflow.

2.1 Extreme Value Theory

Extreme Value Theory [10 11] is concerned with the analysis of extreme values such as maxima, minima or exceedances, instead of the sample mean values of random variables. The theory has been used to model wind speeds, maximum temperatures, floods and risk analysis in finance. There are two types of extreme events. The first type counts the number of occurrences of such events and the second one measures their magnitude. Our work belongs to the second type of extreme events because we are interested in measuring the data value of a variable.

An extreme value is the largest or smallest value in a data set and Extreme Value Theory models these extremes from statistical samples. A sample may consist of data points or values of a random variable. If upper extreme values are observed, then the maximum value from the sample is taken. If lower extreme values are observed, then the minimum value from the sample is taken. For n samples, the extreme values of n maxima and n minima are extracted and statistically analyzed to find extreme value distributions. *Central Limit Theorem* [9] could have been applied if we were interested in mean values rather than the extreme ones. However, extreme values do not fit into the normal distribution even with larger samples. Fortunately, maxima and minima distributions are known to converge to Gumbel distribution [10 11]. The distribution function of Gumbel distribution for maxima is as follows:

$$G(x) = e^{-e^{-\frac{(x-\mu)}{\sigma}}} \quad (1)$$

Here, μ and σ are the location and scale parameters. The location parameter simply shifts the distribution to left or right and the scale parameter stretches it out. The *method of moments estimators* of μ and σ parameters are defined below.

$$\sigma = \frac{s\sqrt{6}}{\pi}, \quad \mu = \bar{x} - \sigma\lambda \quad (2)$$

s and \bar{x} represent the sample mean and standard deviation, respectively and λ denotes Euler's constant (0.5772).

Minima also fit into the Gumbel distribution and their estimators can be computed using the same Equation 2. First, minima are converted into maxima by changing the sign of the minimum values. Then, the Gumbel distribution can be applied to the maxima by computing σ and μ estimators.

2.2 Methodology

Extreme Value Theory can be adapted to bit-width computation of variables by observing maxima and minima values. Variables are assumed to be stored in two's complement number format. The program under observation must be executed with random input samples in order to perform the statistical data analysis of extreme values. A random data sample represents the entire data population and also every sample of the same size has an equal chance to be selected.

The analysis begins by detecting input variables and their bit-widths in the original program. Then, an *input sample* consisting of all data required to run the program is generated randomly. A random input sample always contains valid input data because it is drawn from valid data input population. One must generate a sufficient number of random input samples in order to perform a reliable statistical data analysis. The larger the sample size is, the better the expected distribution fits into the samples. The sufficient number of samples will be determined after the empirical study in *Section 4.2*.

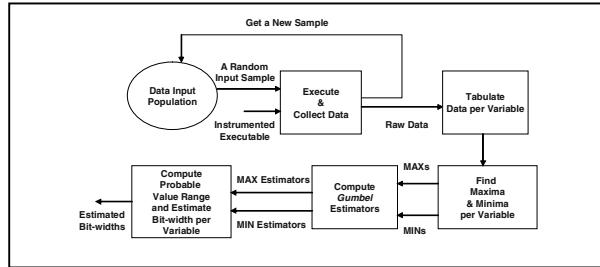


Fig. 1. The bit-width estimation tool

Fig. 1 shows the bit-width estimation tool that is designed to generate the estimated bit-widths using stochastic bit-width approximation. First, n random input samples are generated based on the input variables. Then, each input sample is executed by the *Instrumented Executable* that outputs the values of variables and the values of comparison variables in the conditional statements. As for arrays, the value of an array element is outputted wherever it is assigned or compared but the goal is to find the maximum probable range of any element of the array, rather than the bit-width of individual array elements. The instrumentation details are explained in *Section 3*.

The raw data outputted from the *Execute and Collect Data* stage is collected for tabulation after the program has run all the samples. For each sample run, all possible values for each variable are tabulated by the *Tabulate* stage in a *value table* because more than one value may have been outputted for a variable. After building the value tables, the maximum and minimum values of each variable are selected from each sample and put into *MAX* and *MIN* sample tables by the *Find Maxima and Minima* stage. So, n *MAX* and n *MIN* values are gathered for each variable in the sample tables after running n samples. For each variable, *MAX* and *MIN* sample tables are examined separately to compute Gumbel maxima and minima estimators by the *Compute Gumbel Estimators* stage. Finally, the *Compute Probable Value Range and Estimate Bit-widths* stage determines the probable value range and estimates the bit-width of each variable.

The probable value range of the variable is approximated by estimating *global maximum* and *minimum* values using maxima and minima estimators for a given probability c . The estimated global maximum and minimum values are computed using equation (3). For a random variable X , equation (3) states that X takes values less than or equal to a with probability of c .

$$F(a) = P(X \leq a) = c, \quad 0 < c < 1 \quad (3)$$

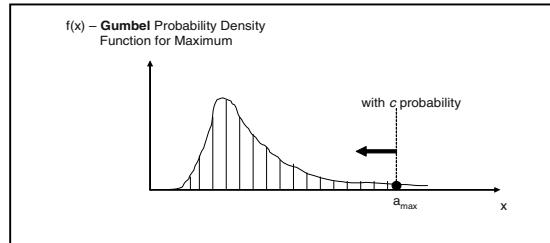


Fig. 2. The Gumbel probability density function for Maximum Values

Fig. 2 shows the Gumbel probability density function and the selection of the estimated maximum value based on a probability c for distribution of maximum values. With c between 0 and 1, we expect that $100c$ percent of the data values will be smaller or equal to a_{max} (i.e. the area under the curve). For example, we expect that 99.9% of x values in the given probability distribution will be smaller than a_{max} if c is 0.999. In other words, there is 0.1% chance (i.e. over/underflow probability) that the variable may be assigned a value larger than the estimated a_{max} . It is clear that the higher c is, the lower the chances of overflow or underflow are. Hence, c probability will be called *in-range probability* of a variable. Similarly, the complementary event of the in-range probability is the *over/underflow probability*, which is represented by f and given by:

$$f = 1 - c \quad (4)$$

We replace the F distribution function in equation (3) with Gumbel distribution as shown in equation (5).

$$P(X \leq a) = e^{-e^{\frac{(a-\mu)}{\sigma}}} = c \quad (5)$$

Solving a from equation (5), we obtain the following equation:

$$a = \mu - \sigma \ln(\ln(\frac{1}{c})) = \mu - \sigma \ln(\ln(\frac{1}{1-f})) \quad (6)$$

Here, a represents the estimated global minimum or maximum. We can estimate a_{min} and a_{max} after calculating estimators (i.e. μ and σ) and selecting a high probability of c or a low probability of f . Hence, the probable value range of a variable V is defined as below:

$$V_{\text{probable value range}} = [a_{\min}, a_{\max}] \quad (7)$$

Equation (7) states that a value that is assigned into V at any time will be within $V_{\text{probable value range}}$ with a probability of c . Finally, the estimated bit-width for variable V can be computed in two's complement number representation with the following equations.

$$V_{\text{bit-width}} = \begin{cases} \lceil \log_2(|a_{\min}|) + 1 \rceil & \text{if } a_{\min} < 0 \text{ and } |a_{\min}| > |a_{\max}| \\ \lceil \log_2(|a_{\max}| + 1) + 1 \rceil & \text{if } a_{\min} < 0 \text{ and } |a_{\max}| > |a_{\min}| \\ \lceil \log_2(a_{\max} + 1) \rceil & \text{if } a_{\min} > 0 \end{cases} \quad (8)$$

If a_{\min} is negative, the variable is a *signed integer*. Otherwise, it is an *unsigned integer*.

3 Experimental Framework

Our experimental compiler infrastructure shown in Fig. 3 is based on the SUIF compiler [15]. The shaded boxes represent the tools we ourselves have implemented within the infrastructure. An application written in C is converted into SUIF Intermediate Representation (SUIF-IR) using the SUIF frontend. After the SUIF frontend optimizations, the IR is sent to the *Instrumentation Tool*. The *Instrumentation Tool* inserts C *printf* library calls to collect the values of assignment and comparison variables, and also variables and arrays that are statically initialized. The instrumented IR is translated into C using the SUIF-to-C translator, and then the instrumented C code is compiled using *gcc* to produce an instrumented executable. Finally, the instrumented executable runs random input samples.

The estimated bit-widths determined by the Bit-width Estimation Tool are sent to the *SUIF-to-Handel-C translator*, a tool implemented in our compiler infrastructure. Handel-C is a high-level Hardware Description Language (HDL) developed by Celoxica [8]. Basically, the SUIF-to-Handel-C translator takes the SUIF-IR code, annotates the estimated bit-width (including signed/unsigned information) to each variable's symbol table entry, and then translates the annotated SUIF-IR code into Handel-C code. The Handel-C compiler compiles the generated Handel-C code and produces an EDIF netlist. Finally, the EDIF netlist is read and customized onto a Xilinx Virtex-II XC2V6000 FPGA [16] by Xilinx Virtex-II EDA tools to produce area and power consumption data.

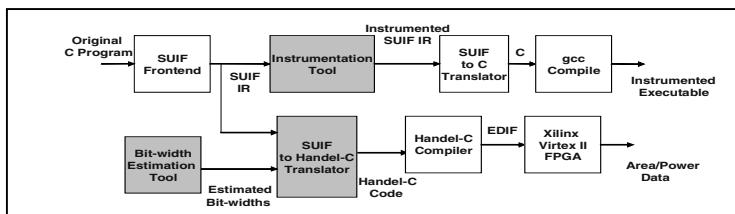


Fig. 3. Our experimental compiler infrastructure

We used nine benchmark programs representing DSP, video, speech and telecommunication applications. Table 1 shows the benchmarks and their descriptions.

Table 1. Benchmarks

Benchmark	Description
adpcm (adp)	PCM to ADPCM speech encoder
matmul (mat)	8-by-8 matrix multiplication
shellsort (she)	sorting algorithm
convolve (conv)	2-dimensional convolution algorithm
FIR	Finite Impulse Response filtering
IDFT	Inverse Discrete Fourier Transform
huffman (huf)	Compression using Huffman encoding
g721	CCITT G.721 ADPCM decoding
viterbi (vit)	Convolutional code decoder using Viterbi algorithm for K=7 with rate=1/2

4 Experimental Statistical Analysis

4.1 Empirical Analysis of Over/Underflow Probabilities

A thorough analysis is needed to select a higher in-range probability c or lower over/underflow probability of f for more accurate statistical data analysis. Fig. 4 shows the change in bit-width in one variable selected from each benchmark by varying f from 0.1 to 10^{-100} .

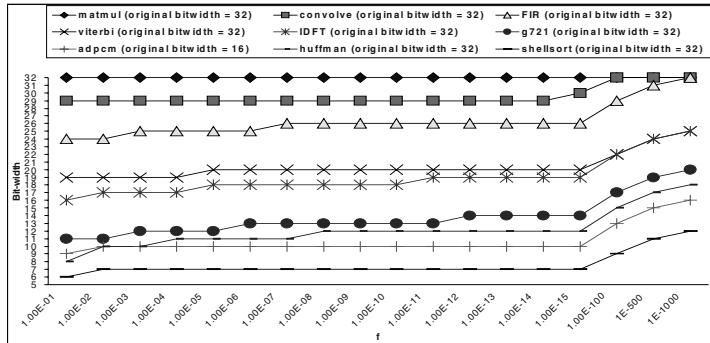


Fig. 4. The variance of bit-width for one variable in each benchmark for a given over/underflow probability

A variable that is highly dependent on input data is selected for demonstration for each benchmark. The original bit-width of each variable is shown in parenthesis on the top of the figure. As f decreases, the estimated global maximum and minimum

both increase if σ is much higher than zero. In this case, the value range of the variable expands. However, the bit-width of the variable in matrix multiplication does not change as the over/underflow probabilities vary. Although its value range changes, this change is not sufficient to increment the bit-width of a variable. This is because the sample standard deviation of the variable is very close to zero as a result of the sample values not deviating significantly from the sample mean. The scale parameter σ , which depends on the sample standard deviation, multiplied by the term $\ln(\ln(1/(1-f)))$ in equation (6) becomes insignificant. Thus, the bit-width stays stable over a range of over/underflow probabilities. In *adpcm*, *convolve* and *fir*, the bit-widths of the variables reach to their original bit-widths when f goes to 10^{-1000} . For the rest of the benchmarks, the bit-widths are still far from their original bit-widths even when f is equal to 10^{-1000} . Note that this probability is so infinitesimally small that the chance of an overflow or underflow is extremely unlikely.

Table 2. The total number of bits of all variables in each benchmark with various over/underflow probabilities for 1000 samples

	adp	FIR	IDFT	mat	huff	shell	conv	g721	vit
Orig.	496	432	784	128	1400	352	864	4616	3480
$f = 10^{-1}$	150	127	341	41	333	122	165	1794	1006
10^{-2}	153	127	343	41	361	123	165	1868	1013
10^{-3}	153	128	343	41	361	123	165	1887	1013
10^{-4}	153	128	343	41	364	123	165	1921	1015
10^{-5}	153	128	346	41	364	123	165	1941	1035
10^{-6}	154	128	348	41	367	123	165	1965	1038
10^{-7}	156	129	352	41	368	123	165	1968	1039
10^{-8}	156	129	356	41	370	123	165	1981	1039
10^{-9}	156	129	358	41	370	123	165	2001	1039
10^{-10}	156	129	359	41	371	123	165	2011	1040
10^{-11}	157	129	361	41	371	123	165	2021	1040
10^{-12}	158	129	362	41	371	123	165	2045	1040
10^{-13}	158	129	362	41	373	123	165	2061	1043
10^{-14}	158	129	364	41	373	123	165	2079	1043
10^{-15}	158	129	364	41	373	123	166	2087	1044
10^{-100}	182	136	405	41	424	125	168	2416	1102
10^{-500}	202	142	440	41	491	127	168	2742	1165
10^{-1000}	212	145	456	41	527	128	168	2888	1194

Table 2 presents the sum of bit-widths in all variables for each benchmark for various over/underflow probabilities. For an array variable, the maximum bit-width of all array elements is selected as the bit-width of the array. The experiments are performed for 1000 samples. The first row in the table shows the total number of bits of all variables in the original program. The bit-widths of only a small set of variables in a benchmark change. For instance, the total number of bits in the *FIR* benchmark is 127 for $f=0.1$ and 128 for $f=0.01$. This one bit change is caused by only one variable in the benchmark and the bit-widths of the rest of variables do not change.

In *matmul*, the total number of bits stays stable over all f values because the standard deviations of the variables in this benchmark are so close to zero that the change

in value ranges is not significant enough to increase the bit-width. On the other hand, *huffman*, *g721* and *viterbi* have so many variables whose standard deviations are relatively large that the value ranges of some variables can expand enough to change one bit as f decreases. When f is equal to 10^{-1000} , the total number of estimated bits in variables is still much less than the total number of variable bits in the original program. The selection of f probability value is a trade-off between the application's over/underflow tolerance and hardware space. If smaller area and low power are of major concern, then a much lower value of f than 10^{-1000} can be selected while increasing the over/underflow probability.

Some applications may have less over/underflow tolerance in some variables than other variables in the program. In this case, a high over/underflow probability can be selected for the variables with less tolerance and a lower one is selected for the others. As a rule of thumb, variables with small standard deviations are assigned higher over/underflow probability and variables with relatively big standard deviations are assigned lower over/underflow probability. It is even possible to select a different probability for each variable in the program if the compiler can estimate the tolerance level for each variable.

Some applications have zero tolerance for overflow or underflow in all variables. For such applications, the bit-widths of the variables whose standard deviations are zero can be estimated by using the stochastic bit-width approximation, and the variables whose standard deviations are bigger than zero can be set to their original bit-widths instead of the approximate bit-widths.

4.2 Sample Size Analysis

We also measure the sample size sensitivity of the stochastic bit-width approximation technique by changing the number of sample size for a selected over/underflow probability of 10^{-15} . 10^{-15} is so small that there is one in a quadrillion chance of over/underflow in a variable. As the sample size increases, the data points fit better into the underlying statistical distribution. Also, the larger the sample size is, the more accurate the estimator parameters of a distribution function.

Table 3 shows the total number of bits in each benchmark for various sample sizes. The change in the sample size effects the Gumbel estimator parameters μ and σ in equation (2) because they are calculated using the sample mean and standard deviation. Recall that the estimated global *MAX* and *MIN* values depend on these estimator parameters (equation (6)). Although the probable value range expands as the estimator parameters change with the sample size, the bit-width of variables is wide enough to hold the expanded value range. Hence, the total number of bits in *matmul* and *shellsort* does not vary as the sample size increases. However, the total number of bits in the other benchmarks changes as the sample size increases but stabilizes at different sample numbers denoted by the shaded cells. The stabilization samples numbers are 150, 500, 150, 200, 250, 650 and 150 for *adpcm*, *FIR*, *IDFT*, *huffman*, *convolve*, *g721* and *viterbi*, respectively. Thus, 650 samples are sufficient to perform a reliable statistical analysis for all the benchmarks.

Table 3. The total number of bits in all variables in each benchmark for various random input data sample size with $f=10^{-15}$

Sample Size	adp	FIR	IDFT	mat	huff	shell	conv	g721	vit
100	159	130	365	41	372	123	166	2084	1046
150	158	130	364	41	372	123	166	2092	1044
200	158	130	364	41	373	123	166	2089	1044
250	158	130	364	41	373	123	165	2083	1044
300	158	130	364	41	373	123	165	2092	1044
350	158	130	364	41	373	123	165	2091	1044
400	158	129	364	41	373	123	165	2091	1044
450	158	130	364	41	373	123	165	2090	1044
500	158	129	364	41	373	123	165	2090	1044
550	158	129	364	41	373	123	165	2084	1044
600	158	129	364	41	373	123	165	2086	1044
650	158	129	364	41	373	123	165	2087	1044
700	158	129	364	41	373	123	165	2087	1044
750	158	129	364	41	373	123	165	2087	1044
800	158	129	364	41	373	123	165	2087	1044
850	158	129	364	41	373	123	165	2087	1044
900	158	129	364	41	373	123	165	2087	1044
950	158	129	364	41	373	123	165	2087	1044
1000	158	129	364	41	373	123	165	2087	1044

4.3 Testing Over/Underflow Probabilities by Monte Carlo Simulation

We use the Monte Carlo simulation to verify the correctness of the selected over/underflow probability. Basically, the Monte Carlo simulation creates several input sets for a problem by selecting random values. As the number of input sets increases, an approximate stochastic solution to the problem can be reached.

For each benchmark, we first compute the bit-widths of variables by estimating the global maximum and minimum values with an over/underflow probability for 650 samples. Then, we generate 10,000 random input samples, execute the benchmark with each input sample and observe the values assigned to each variable for potential overflows or underflows for the estimated bit-width assigned to the variable. Like this, we can obtain the potential number of overflows or underflows for each variable over 10,000 samples. Once the number of over/underflows is known, the *empirical over/underflow probability* or ef can be calculated for the variable that encountered over/underflow. ef is calculated by dividing the number of over/underflows by 10,000 samples for each variable. Then, ef is compared with the selected over/underflow probability f for the variable to see how close the empirical over/underflow probability is to the compile-time selected over/underflow probability.

We expect ef to be very near to f . In theory, a maximum of 100 over/underflows may occur in a variable out of 10,000 input samples when f is equal to 10^{-2} . Similarly, at most 1 over/underflow may occur out of 10,000 input samples when $f=10^{-4}$. In other words, a variable may overflow or underflow at most for 100 samples when $f=10^{-2}$, and at most for 1 sample when $f=10^{-4}$ out of 10,000 samples.

Table 4 shows the over/underflow status of each benchmark after performing Monte Carlo simulation with 10^{-2} and 10^{-4} selected over/underflow probabilities for 10,000 random samples. Only two benchmarks show over/underflows in a few variables. The variables that encounter over/underflows and their empirical over/underflow probabilities are shown in Table 5. Only ten variables in *g721* encounter overflow or underflow, and only one variable in *FIR* encounters overflow for 10,000 random input samples. As seen in the table, ef values for all variables in both benchmarks are very close to 10^{-2} when f equals to 10^{-2} , and no over/underflow is observed in any variable in both benchmarks when $f=10^{-4}$. These results strongly support our argument that the chance of an overflow or underflow in a variable can be made improbable by selecting a very small value of probability f .

Table 4. Over/underflow status of each benchmark after Monte Carlo simulation for 10,000 random samples

Benchmark	Over/Underflow Status
g721	Yes
FIR	Yes
all others	No

Table 5. Empirical over/underflow probabilities in over/underflowed variables for *g721* and *FIR*

g721		<i>f</i>	
Over/underflowed variables		10^{-2}	10^{-4}
<i>ef</i> for Var 1		0.015	0
<i>ef</i> for Var 2		0.0002	0
<i>ef</i> for Var 3		0.0005	0
<i>ef</i> for Var 4		0.0002	0
<i>ef</i> for Var 5		0.0002	0
<i>ef</i> for Var 6		0.002	0
<i>ef</i> for Var 7		0.001	0
<i>ef</i> for Var 8		0.0001	0
<i>ef</i> for Var 9		0.0001	0
<i>ef</i> for Var 10		0.0002	0
FIR		<i>f</i>	
Overflowed variable		10^{-2}	10^{-4}
<i>ef</i> for Var 1		0.0007	0

4.4 Analysis of Bit-Width Reduction

Fig. 5 shows the percentage reduction in the total number of bits in variables for each benchmark using our technique, with respect to the originally declared bit-widths of the benchmark. We run all the benchmarks for 650 random samples with f equal to 10^{-15} . For array variables, the bit-width of an array is considered instead of the bit-width of each array element.

As seen in the graph, a great amount of bit-width reduction occurs in *adpcm*, *FIR*, *huffman*, *convolve*, *g721* and *viterbi* by 59%, 60%, 51%, 34%, 46% and 52% respectively. These benchmarks have large arrays declared as 32 bits in the original program

but their estimated bit-widths are much less than 32 bits. For instance, *huffman* has an array of structures with 256 elements and the size of each structure is 80 bits. The bit-width of this structure is estimated to be only 40 bits using stochastic bit-width approximation. The percentage reductions are relatively small in *IDFT*, *matmul* and *shellsort* by 2%, 4% and 5%, respectively. These are the kind of programs where only the bit-widths of loop induction variables and some temporary variables can be reduced. The bit-widths of array variables stay the same as the original program. The overall bit-saving in variables with respect to the original bit-widths is an average of 35% for the nine benchmarks.

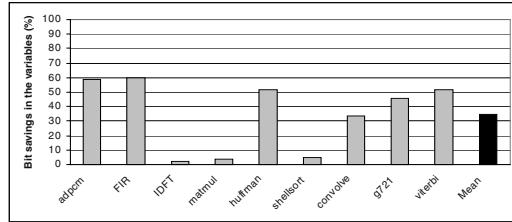


Fig. 5. The percentage of reduction in the total number of bits

5 Experimental Implementation Analysis

5.1 FPGA Area Results

We expect that bit-width reduction in operands will lead to reduction in FPGA circuitry such as narrower multiplexers, functional units, and shorter routes between the logic blocks. Particularly, bit savings in arrays may reduce the number of look-up tables (LUT) dramatically. Fig. 6 shows the percentage reduction in the number of LUTs in comparison to the implementation of the original program.

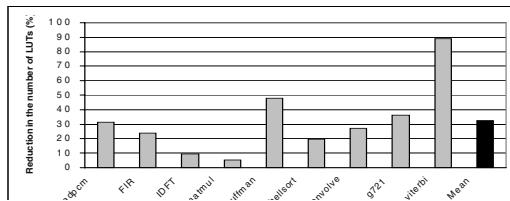


Fig. 6. The percentage reduction in the number of the look-up tables (LUTs)

The largest reduction in terms of LUTs occurs in *viterbi* by 89%. This is caused by not only the bit-width savings in arrays but also by a large number of narrower functional units where these array elements are used as source and destination operands. Although bit savings in arrays in Fig. 5 were huge in *adpcm*, *FIR*, *huffman*, *convolve*

and *g721*, array elements are not used in the computations as aggressively as in *viterbi*. The smallest area reduction occurs in *matmul* by 5.5%. This reduction comes from narrower comparators of the loop index variables whose bit-widths are reduced in the nested loops. As a result, we can attain an average of 32% LUT reduction in the FPGA area for the nine benchmarks.

5.2 FPGA Power Consumption Results

We perform two different power measurements: 1) the design power and 2) the total power consumptions. The design power consumption is the sum of the logic block power and the signal power. The logic block power is power consumed by the logic gates, and the signal power is power consumed by the signals and nets between the logic gates. The total power consumption is the sum of quiescent, logic block, signal power, and power consumed by input/output buffers and clocks. We measure power consumption using the *Xilinx XPower* tool that can estimate power consumption by $\pm 10\%$ error after placing and routing each benchmark on the FPGA. Each benchmark (both original and bit-reduced versions) is fed by a 100MHZ clock under 1.5V voltage source with a toggle rate of 100% or $\frac{1}{2}$ the clock frequency. The toggle rate is the rate at which a logic gate switches.

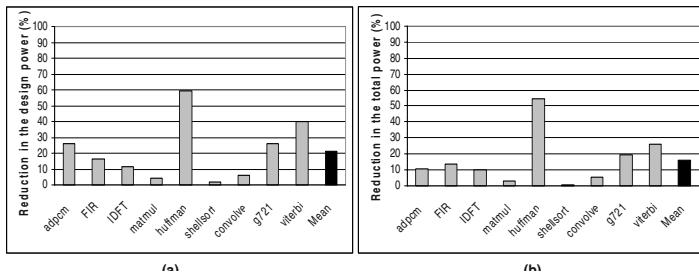


Fig. 7. The percentage reduction in power consumption

Fig. 7 shows the percentage reduction in the design and total power consumption with respect to power consumed by the implementation of the original program. Reduction in design power consumption as shown in Fig. 7a is consistent with reduction in the number of LUTs except for the fact that the largest power reduction is in *huffman*, a saving of 59%. Although *viterbi* has the largest reduction in area, this does not proportionally reflect on the design power due to the power consumed by the signals and nets between the gates. Thus, the signal power consumed by signals and nets takes over and diminishes the gain benefited from logic block power reduction. A similar effect can be observed in *shellsort*, which has the lowest design power reduction at 1.6%, where the signal power dominates the logic power. The other benchmarks show power reductions at a moderate level. The overall design power reduction is an average of 21% for the nine benchmarks. Fig. 7b shows reductions in total power consumption, which is an average of 16% for the nine benchmarks.

6 Related Work

Value range and precision of variables have been analyzed in the context of converting the floating-point representation to the fixed-point representation for DSP applications [18 19]. Also, a similar work has been performed to customize floating-point precision requirements for DSP applications [17]. As for bit-width analysis for integer variables, there are two categories of previous work: static and dynamic techniques. Static techniques [1 2 4 5 6 7] use iterative backward and forward data flow analysis to infer bit-widths and detect unused bits by using program constants, loop trip counts, array bounds and masking operations. These techniques will assume the worst possible bit-widths in the absence of such static inference hints in the program. Also, they have to be conservative if program variables tightly depend on inputs, particularly with *while* loops whose loop trip counts are not known at compile time. Whereas, our technique can not only accurately estimate bit-widths of variables with static inference hints, but also can approximate bit-widths of the highly input-dependent variables without overestimation. Stefanovi *et al.* [3] introduces a dynamic technique that constructs a data-flow graph and places instructions, including their input and output operands into the graph while running the program. Their goal is to infer unused bit positions of the analyzed values in the general-purpose applications during the execution of the program. However, our technique eliminates the bias of using a single input by generating a large number of random input samples.

7 Conclusion

We have presented a novel stochastic bit-width approximation technique using *Extreme Value Theory* with statistical random sampling to reduce area and power consumption for customizable processors. Stochastic bit-width approximation estimates the bit-widths of all variables in an embedded application with a finite overflow or underflow probability. It is possible to reduce custom hardware space and circuit power dramatically by tailoring the over/underflow probability for all variables or even by selecting a different probability for each variable in the application.

Our statistical empirical results have showed that the chance of an over/underflow in a variable can be made extremely unlikely by selecting an infinitesimally small probability value. The stochastic bit-width approximation technique can reduce the total number of bits in programs by an average of 35% for the nine embedded benchmarks. This observation is also supported by the actual customization of the benchmarks on the FPGA chip. Overall, the FPGA area can be reduced by 32% in terms of the number of LUTs. The overall reduction in the design power consumption and in the total power consumption is found to be 21% and 16%, respectively.

References

1. M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation", *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, BC, June 2000.
2. R. Razdan, and M. D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", in *Proc. 27th Ann. Int'l Symp. Microarchitecture*, San Jose, CA, Dec. 1994.
3. D. Stefanović and M. Martonosi, "On Availability of Bit-narrow Operations in General-purpose Applications", *The 10th International Conference on Field Programmable Logic and Applications*, Aug. 2000.
4. M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein, "BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations", *Proceedings of the 6th International Euro-Par Conference*, Munich, Germany, Aug. 2000.
5. S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, "Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators", *HPL-2001-209 Technical Report*, Aug. 2001.
6. Y. Cao, and H. Yasuura, "Quality-Driven Design by Bitwidth Optimization for Video Applications", *Proc. of IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC'03)*, Jan. 2003.
7. R. Gupta, E. Mehofer, and Y. Zhang, "A Representation for Bit Section Based Analysis and Optimization", *International Conference on Compiler Construction, LNCS 2304, Springer Verlag*, Grenoble, France, Apr. 2002.
8. Celoxica, Handel-C Language Reference Manual, Version 3.1, 2002.
9. C. H. Brase, and C. P. Brase, "Understanding Basic Statistics", *Houghton Mifflin Company*, 2nd Edition, 2001.
10. R. R. Kinnison, "Applied Extreme Value Statistics", *Macmillan Publishing Company*, New York, 1985.
11. R. D. Reiss, and M. Thomas, "Statistical Analysis of Extreme Values", *Birkhäuser Verlag*, Basel, Switzerland, 1997.
12. D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and Compiling Applications for RaPiD", *Field-Programmable Custom Computing Machines*, 1998.
13. J. Frigo, M. Gokhale, and D. Lavenier "Evaluation of the Streams-C C-to-FPGA Compiler: An Application Perspective", *9th ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, February, 2001.
14. B. A. Draper, A. P. W. Böhm, J. Hammes, W. Najjar, J. R. Beveridge, C. Ross, M. Chawathe, M. Desai, and J. Bins, "Compiling SA-C Programs to FPGAs: Performance Results", *International Conference on Vision Systems*, Vancouver, July, 2001.
15. R. P. Wilson, R. S. French, C. S. Wilson, S. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers", *Technical Report*, Computer Systems Laboratory, Stanford University, CA, USA, 1994.
16. Xilinx, *Xilinx Virtex-II Architecture Manual*, Sep. 2002.
17. C. F. Fang, R. Rutenbar, M. Peuschel, and T. Chen, "Toward Efficient Static Analysis of Finite-Precision Effects in DSP Applications via Affine Arithmetic Modeling", *Design Automation Conference*, 2003.
18. S. Kim, K. Kum, and W. Sung "Fixed-point Optimization Utility for C and C++ Based Digital Signal Processing Programs", *IEEE Trans. on Circuits and Systems*, Vol. 45, No. 11, Nov. 1998.
19. M. Willems, V. Bürgens, H. Keding, T. Grötker, and H. Meyr, "System Level Fixed-point Design Based on An Interpolative Approach", *Proc. 34th Design Automation Conference*, June 1997.

Using Multiple Memory Access Instructions for Reducing Code Size

Neil Johnson and Alan Mycroft

Computer Laboratory, University of Cambridge
William Gates Building, JJ Thompson Avenue,
Cambridge, CB3 0FD, UK
`{Neil.Johnson,Alan.Mycroft}@cl.cam.ac.uk`

Abstract. An important issue in embedded systems design is the size of programs. As computing devices decrease in size, yet with more and more functions, better code size optimizations are in greater demand. For an embedded RISC processor, where the use of compact instructions (*e.g.*, the ARM Thumb) restricts the number of accessible registers at the expense of a potential increase in spill code, a significant proportion of instructions load or store to memory.

In this paper we present a new technique which both identifies sequences of single load and store instructions for combining into multiple load and store instructions, and guides the layout of function stack frames, global storage and register allocation, previously only seemingly done by opportunistic optimization. We implement this in our SOLVEMMA algorithm, similar to Liao’s *Simple Offset Assignment* algorithm. We implement our algorithm within the Value State Dependence Graph framework, describe how the algorithm is specialized for specific processors, and use the ARM Thumb as a concrete example for analysis.

1 Introduction

With increasing numbers of compact and mobile computing devices, such as mobile phones, wrist-watches, and automotive systems, comes a growing need for more effective code-compacting optimizers. One potentially profitable route to compact code is through the effective use of Multiple Memory Access (MMA) instructions. These include multiple loads and stores where a set of registers (typically defined by some range or bitmap representation) are loaded from, or stored to, successive words in memory. Their compactness comes from expressing a (potentially large) set of registers with (comparatively) few instruction-word bits; for example, the ARM LDM instruction uses only sixteen bits to encode up to sixteen register loads (*cf.* 512 bits without the use of the LDM instruction).

The ARM Thumb [15] executes a compact version of the ARM instructions. A hardware translator expands, at runtime, the 16-bit Thumb instructions into 32-bit ARM instructions. A similar approach is taken in the MIPS16 embedded processor [9]. A disadvantage of this approach is that fewer instruction bits are available to specify registers. This restriction artificially starves the register

allocator, resulting in more register spill code (providing more potential sources of MMA optimization, which is poorly done in existing compilers).

The “*load-store*” nature of a RISC architecture also gives rise to many explicit memory access instructions. In contrast, a CISC machine with memory-access-with-operation instructions can achieve better code size by combining a load or store operation (especially where the address offset is small) with an arithmetic operation (*e.g.* “`add EAX,DS:ESI [EBX*4]+Offset`” on the Intel x86 [1]). Restricting the number of accessible registers, as described above, increases the number of memory access instructions: the *adpcm* benchmark from MediaBench [11] generates approximately 35% more memory access instructions for the ARM Thumb than for the ARM.

However, the state of the art in commercial compilers appears¹ to be based on opportunistic peephole-style optimization. The GCC compiler also takes an ad-hoc approach to MMA optimization².

This paper explores the use of MMA instructions as a means of compacting code. Our SOLVEMMA algorithm identifies profitable memory access sequences for combining into MMA instructions, and selects stack frame layouts that facilitate multiple reads and writes of local variables. Interestingly, it turns out that array accesses and local variable accesses are best treated separately.

We describe SOLVEMMA as applied to the Control Flow Graph, and then show how the less-constrained Value State Dependence Graph provides greater opportunities for merging multiple load and store instructions. Finally, we specialize this to the ARM Thumb MMA instructions, and apply our algorithm to a selection of embedded software benchmarks.

1.1 Related Work

As remarked above there is no directly related work on MMA optimization for general purpose registers. However, a related area of research is that of optimizing address computation code for Digital Signal Processors (DSP).

For architectures with no register-plus-offset addressing modes, such as many DSPs, over half of the instructions in a typical program are spent computing addresses and accessing memory [16]. The problem of generating optimal address-computing code has been formulated as the Simple Offset Assignment (SOA) problem, first studied by Bartley [4] and Liao *et al* [13], the latter formulating the SOA problem for a single address register, and then extended to the General Offset Assignment (GOA) problem for k address registers. Liao *et al* also show that SOA (and GOA) are NP-hard, reducing the problem to the Hamiltonian path problem. Their approximating heuristic is similar to Kruskal’s maximum spanning tree algorithm.

Rao and Pande [14] generalize SOA for optimizing expression trees to minimize address computation code. An improved heuristic for SOA and GOA was

¹ Through literature and private communication with industrial compiler developers.

² For example, the GCC compiler uses *hard registers* to enforce a particular register assignment within the RTL intermediate form.

proposed by Leupers and Marwedel [12] with a tie-breaking heuristic for SOA and a variable partitioning strategy for GOA.

DSP-like architectures (*e.g.*, Intel MMX, PowerPC AltiVec and Sun VIS) include MMA-like block loads and stores. However, these instructions are limited to fixed block loads and stores to special data or vector registers, not general purpose registers. A related area to MMA optimization is that of *SIMD Within A Register* (SWAR) [6]. The work presented here considers only word-sized variables; applying the same algorithm to sub-word-sized variables would achieve additional reductions in code size (*e.g.*, combining four byte-sized loads into a single word load).

Koseki *et al* consider the problem of colouring a CFG (where the order of instructions is fixed) in the presence of instructions which have particular preferences for register assignment [10]. They suggest using these preferences to guide register assignment to enable the use of MMA instructions. We differ from their work in two ways: (1) because the VSDG underspecifies the ordering of instructions in a graph³ we can consider combining loads and stores that are not adjacent to each other in the CFG into provisional MMA nodes; and (2) we use the Access Graph during target code generation to bias the layout of the stack frame for spilled and local variables.

All of the above approaches have been applied to the Control Flow Graph [3], which precisely specifies the sequence of memory access instructions. By contrast, the Value State Dependence Graph [8] under-specifies the ordering of most operations, only specifying *sufficient* partial ordering to maintain the I/O semantics (memory reads and writes) of the original program, giving more opportunities for optimization.

1.2 Paper Structure

The rest of this paper is structured as follows. In Section 2 we summarize Liao *et al*'s SOA algorithm. Section 3 describes our new SOLVEMMA algorithm in the context of the Control Flow Graph, with Section 4 enhancing the MMA algorithm for the Value State Dependence Graph. In Section 6 we apply SOLVEMMA, implemented within our VSDG compiler framework, to a number of illustrative examples, and close with concluding remarks (Section 7).

2 Simple Offset Assignment

The Simple Offset Assignment (SOA) algorithm [13] rearranges local variables within a function's stack frame in order to minimize address computations. The target architecture is assumed to have a single address register with word-oriented auto-increment/decrement addressing modes.

³ We showed previously [8] that the special case of a VSDG with *enough* serializing edges to enforce a linear order corresponds to a CFG.

The input to the algorithm is an instruction-scheduled and register-allocated program, *i.e.*, a Control Flow Graph (CFG) [3], with a fixed memory access sequence. The SOLVESOA algorithm constructs an *Access Graph* (V, E), an undirected graph with vertices V and edges E . Vertices correspond to variables, and there is an edge $e = (p, q)$ between vertices p and q with weight $w(e)$ if there are $w(e)$ adjacent accesses to variables p and q .

For our purposes we say that a *covering* of a graph is a subset of its edges, and recall that a path is an alternating sequence of vertices and edges, with each edge connecting its adjacent vertices, and there are no cycles in the path.

The algorithm then *covers* the Access Graph with one or more maximally-weighted disjoint *paths*. Each path specifies an ordering of the variables in the path on the stack, thereby minimizing the number of address computations through the use of auto-increment/decrement addressing. The number of address computations is then given by the sum of the weights of the uncovered edges.

Finding an optimal path covering is a formulation of the Maximum Weight Path Covering (MWPC) problem, which has been shown [13] to be in NP. Liao *et al* propose a greedy algorithm (similar to Kruskal's maximum spanning tree algorithm [2]) which iteratively chooses the edge with the greatest weight to add to the path while preserving the properties of the path (if two or more edges have the same weight the algorithm non-deterministically chooses one of them). It terminates when either no further edges can be added to the solution, or there are no more edges. An example of SOA is shown in Figure 1.

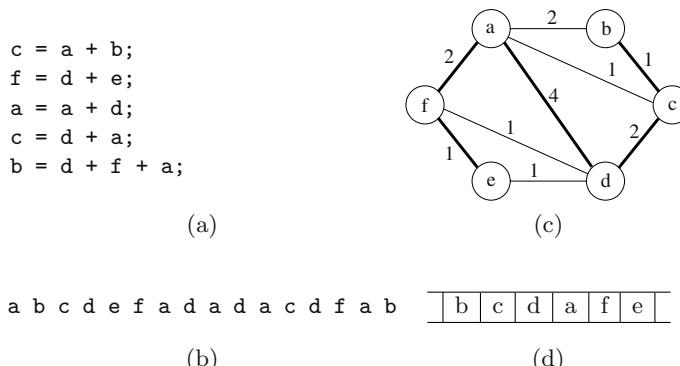


Fig. 1. Example illustrating SOA. In (a) is a short code sequence, accessing variables $a-f$. The access sequence (b) describes the sequence of read or write accesses to the variables (for illustration we assume variables in expressions are accessed in left-to-right order). This leads to the Access Graph shown in (c), where the edge weights correspond to the number of times a given sequence occurs in (b). We then cover the graph using the MWPC algorithm, with covered edges shown in bold. The result is the stack frame layout shown in (d).

Computing this approximate MWPC can be done in $O(|E| \log |E| + |L|)$ time, where $|E|$ is the number of edges in the Access Graph, and $|L|$ the number of variable accesses. Liao *et al* showed that for large programs the Access Graphs are generally quite sparse.

The General Offset Assignment (GOA) algorithm extends SOA to consider k address registers (consider SOA as a special case of GOA, with $k = 1$). The SOLVEGOA algorithm grows an Access Graph, adding vertices to it if adding an additional address register to the graph would likely contribute the greatest reduction in cost. The decision of *which* variables to address is left as a heuristic.

3 Multiple Memory Access on the Control Flow Graph

In this section we describe our SOLVEMMA algorithm in the context of the Control Flow Graph (CFG) [3]. Similar to Liao *et al*'s SOLVESOA algorithm, we construct an *Access Graph* from the input program and then find a covering which maximizes provisional MMA instructions and biases register assignment to enable as many of these as possible to become real.

The SOLVEMMA algorithm is different to SOLVESOA in two distinct ways. Firstly, the goal of SOLVEMMA is to identify groups of loads or stores that can be profitably combined into MMA instructions. And secondly, SOLVEMMA is applied *before* instruction scheduling, when it is easier to combine loads and stores into provisional MMA instructions and to give hints to the register assignment phase.

3.1 Generic MMA Instructions

For discussion purposes we use two generic MMA instructions—**LDM** and **STM**—for load-multiple and store-multiple respectively. The format of the LDM (and STM) instruction is:

```
LDM    Addr, { reglist }
```

where **Addr** specifies some address computation (*e.g.*, register or register+offset) containing the address from which the first word is read, and the registers to be loaded are specified in **reglist**. As in the ARM instructions, the only constraint on the list of registers is that they must be sorted in increasing numerical order. See Section 5 for specific details of addressing modes and other target-specific details.

The SOLVEMMA algorithm is applied twice to the flow graph. The first pass transforms global variables whose address and access ordering is fixed in the CFG⁴, while the second pass transforms local and spill variables, whose addresses are not fixed, and can in more cases be re-ordered.

⁴ But note Section 4 where using the VSDG instead allows only essential (programmer-required) ordering to be fixed.

3.2 Access Graph and Access Paths

We define *Access Graph* and *Access Paths* as a framework in which to formulate the problem and the SOLVEMMA algorithm.

Definition 1. Let $\alpha(p)$ be the set of instructions which access variable p and $op(i)$ be the operation (load or store) of instruction i . Then the Access Graph is a weighted directed graph $AG = (V, E)$, consisting of vertices V , one for each variable, and edges $E \subseteq V \times V$ such that there exists an edge $(p, q) \in E$ between vertices p and q iff (1) $i \in \alpha(p)$ and $j \in \alpha(q)$ are in the same basic block; (2) $op(i) = op(j)$; (3) j is scheduled after i ; and (4) j is not data-dependent on i .

Vertices in V are tagged with the direction property $dir \in \{UND, HEAD, TAIL\}$ to mark *undecided*, *head* and *tail* vertices respectively. Initially, all nodes are marked *UND*. For some access sequences the direction of accesses must be enforced (e.g., accesses to memory-mapped hardware registers), while in others the sequence is equally valid if traversed in either direction. The direction property marks explicit directions, while deferring the final direction of undefined access sequences to the covering algorithm. Our use of a directed Access Graph is in contrast to SOA's undirected Access Graph.

Definition 2. The Weight $w(e)$ of an edge $e = (p, q) \in E$ is given by $w(e) = |\{(i, j) | i \in \alpha(p), j \in \alpha(q), i \text{ is before } j \text{ in the same basic block, and only arithmetic operations between them}\}|$.

It is possible that some pairs of vertices are accessed in both combinations (p, q) and (q, p) . Some of these edge pairs are likely to have differing weights, due to explicit preferences in the program (e.g., a pair of stores). Such edges are called *unbalanced* edges:

Definition 3. An Unbalanced Edge is an edge $e = (p, q) \in E$ where either there is an opposite edge $e' = (q, p) \in E$ such that $w(e) \neq w(e')$, or where such an opposite edge does not exist.

The definition of the access path follows from the above definition of the Access Graph.

Definition 4. An Access Path $C = (V_C \subseteq V, E_C \subseteq E)$ where $|E_C| = |V_C| - 1$, is a sequence of vertices $\{v_1, v_2, \dots, v_m\}$ where $(v_i, v_{i+1}) \in E_C$ and no v_i appears more than once in the sequence.

An Access Graph can be covered by two or more *disjoint* access paths:

Definition 5. Two paths $C_A = (V_A, E_A)$ and $C_B = (V_B, E_B)$ are disjoint if $V_A \cap V_B = \emptyset$.

Note that SOLVESOA does not consider the type of access *operation*. In contrast, SOLVEMMA combines instructions of the same operation, so path covering must be sensitive to access operations. Consider the access sequence of three local variables:

$$a_r b_r c_w b_r a_r b_w$$

where the subscript r denotes read and w denotes write. It may be that a , b and c are placed in contiguous locations, but the write to c prevents the construction of a single MMA access path $\{a, b, c\}$, whereas SOA would place all three in a path. When SOLVEMMA is applied to the VSDG (Section 4), which would not specify the order of the reads of a and b , then either $\{a, b\}$ or $\{b, a\}$ can be normalised to the other, subject to any restrictions imposed by target-specific MMA instructions.

3.3 Construction of the Access Graph

The Access Graph is constructed in a single pass over the input program, as directed by Definition 3.2. The first two criteria restrict merging to access instructions of the same operation (load or store) and that both are within the same basic block (this maintains the ordering of memory accesses in the CFG). The third criterion ensures that there are no intervening load or store instructions that might interfere with one or the other instructions (*e.g.*, for two loads, an intervening store to the same address as the second load).

Vertices in the Access Graph represent variables whose address can be statically determinable and are guaranteed not to alias with any other vertex⁵. While this restriction may seem harsh, it supports global variables, local (stack-based) variables, including compiler-generated temporaries, and register-plus-constant-offset (indexed array) addressing. In these cases we can statically determine if two memory accesses do not alias to the same address.

3.4 SOLVEMMA and Maximum Weight Path Covering

We follow the approach taken in SOLVESOA of a greedy MWPC algorithm (Section 2). The algorithm (Figure 2) is greedy in that on each iteration it selects the edge with the greatest weight (if two or more edges have the same weight the algorithm non-deterministically chooses one of them).

In the following definitions let AG be an Access Graph (Definition 3.2) with vertices V and edges E .

Definition 6. A partial disjoint path cover of a weighted Access Graph AG is a subgraph $C = (V' \subseteq V, E' \subseteq E)$ of AG such that $\forall v \in V', \text{degree}(v) \leq 2$ and there are no cycles in C ; an orphan vertex is a vertex $v \in V \setminus V'$.

⁵ Aliasing may produce loops in the access path. For instance, consider the path $a-b-c-d$; if c is an alias for b , after addresses have been assigned the resulting offset sequence would be, say, 0-4-4-8. MMA instructions can only access contiguous memory locations, neither skipping nor repeating addresses, and thus this sequence of accesses is not possible.

Applying the SOLVEMMA algorithm to the Access Graph produces a partial covering of the graph. It is partial in that some vertices in the graph may not be covered by an access path. An orphan vertex identifies a variable that cannot be profitably accessed with an MMA instruction.

```

// Take program  $P$ , construct its Access Graph  $(V, E)$  return a covering
// Access Graph  $(V', E')$ .
1. procedure SOLVEMMA (  $P:\text{CFG}$  ): AccessGraph
2.    $(V, E) \leftarrow \text{CONSTRUCTACCESSGRAPH}(P);$ 
3.    $E_{\text{sort}} \leftarrow \text{SORTDESCENDINGORDER}(E); \quad // \text{Sort } E \text{ by weight.}$ 
4.    $V' \leftarrow V, E' \leftarrow \emptyset;$ 
5.   while  $|E'| < |V| - 1$  and  $E_{\text{sort}} \neq \emptyset$  do
6.      $e \leftarrow \text{greatest edge in } E_{\text{sort}};$ 
7.      $E_{\text{sort}} \leftarrow E_{\text{sort}} - \{e\};$ 
8.     if  $e$  does not cause any vertex in  $V'$  to have degree  $> 2$  and
9.        $e$  does not cause a cycle in  $E'$  and
10.       $\text{head}(e).\text{dir} \in \{\text{UND}, \text{TAIL}\}$  and
11.       $\text{tail}(e).\text{dir} \in \{\text{UND}, \text{HEAD}\}$  then
12.         $E' \leftarrow E' + \{e\};$ 
13.         $e' \leftarrow \text{reverse-edge} \in E \text{ of } e;$ 
14.        if  $e' = \emptyset$  or  $\text{weight}(e') \neq \text{weight}(e)$  then
15.          walk from  $\text{head}(e) \in V'$  marking UND vertices as HEAD;
16.          walk from  $\text{tail}(e) \in V'$  marking UND vertices as TAIL;
17.        endif
18.        else
19.          discard  $e$ ;
20.        endif
21.      endwhile
22.      return  $(V', E');$ 
23. endproc

```

Fig. 2. The SolveMMA algorithm. The steps additional to the SolveSOA algorithm (marked with a ‘+’) are lines 10 and 11, which ensure that we never add edges that violate the direction of directed paths, and lines 13–17 which convert undirected paths into directed paths if e is an unbalanced edge (Definition 3.2).

The first step of the SOLVEMMA algorithm (Figure 2) constructs the Access Graph (V, E) from the input program P . The set of edges E is then sorted in descending order by weight, a step which simplifies the operation of the algorithm. We initialise the output Access Graph (V', E') with $V' = V$, and no edges.

The main body of the algorithm (lines 5–21) processes each edge, e , in E_{sort} in turn until either there are just enough edges in the solution to form one long path (at which point we can add no further edges to E' that would satisfy the criteria on lines 8 or 9), or there are no more edges in E_{sort} . We remove e from

E_{sort} and then decide whether it should be added to C . To do so, it must meet the following four criteria:

1. the edge must not cause a cycle in E' ;
2. the edge must not cause any vertex in V' to have degree > 2 , i.e., no edge can connect to an internal vertex within a path;
3. the head of the edge can only connect to a vertex that is either the tail of a directed path, or the end of an undirected path; and
4. the tail of the edge can only connect to a vertex that is either the head of a directed path, or the end of an undirected path.

If all four criteria are satisfied we add e to E' .

Initially, all vertices (and hence paths constructed from them) are marked *UNDecided*, reflecting no preference in the access order. However, it is very likely that in (V, E) there will be some sequences of access that are more favourable than others (e.g., if there was one instance of (p, q) and two instances of (q, p)). This is reflected in a difference between the weights of the edges (p, q) and (q, p) . Indeed, there may not even be a matching reverse edge (line 13).

The remaining lines, 15 and 16, handle the case of e being unbalanced, marking all the vertices from the head of e with *HEAD*, and all the vertices from the tail of e with *TAIL*. Note that this can happen at most once per path, as any subsequent addition to the path must respect its direction (lines 10 and 11). This simple heuristic works well in practice and has low runtime cost.

3.5 The Phase Order Problem

An important problem encountered by compiler designers is the *phase ordering* problem, which can be phrased as “*in which order does one schedule two (or more) phases to give the best target code?*”. Many phases are very antagonistic towards each other; two examples being code motion (which may increase register pressure) and register allocation (which places additional dependencies between instructions, artificially constraining code motion).

If SOLVEMMA is applied before register allocation then any subsequent spill code generated by the register allocator would not be considered by SOLVEMMA, and additional constraints specified by the semantics of a given target’s MMA instructions would be imposed on the operation of the register allocator. If SOLVEMMA is applied after the instruction scheduling phase then the scheduler might construct a sequence that prevents several memory access instructions becoming a single MMA instruction.

3.6 Scheduling SOLVEMMA within a Compiler

We now describe the order in which MMA optimizations are applied to a CFG in our implementation. While this order does not produce optimal code for all programs, early indications are that this order is generally good.

The first pass of *MMA Optimization* takes in a scheduled CFG, where the order of instructions (especially loads and stores) is fixed. At this stage the only

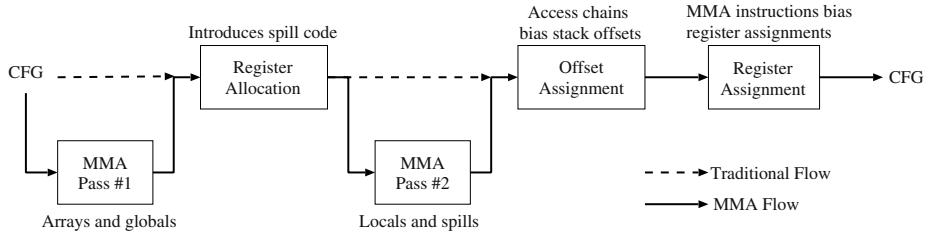


Fig. 3. Scheduling MMA optimization with other compiler phases.

memory accesses are to global variables (fixed addresses) and local arrays and structs (fixed offsets from a base address)⁶. The access graph constructed during this pass identifies *provisional* MMA instructions, albeit in a fashion which we can undo later.

The second phase is *Register Allocation*. This inserts spill code (compiler-generated loads and stores to temporaries placed on the stack) into the CFG where there are insufficient physical registers to be able to colour a register.

The third phase is another *MMA Optimization* pass, but now concerned with loads and stores introduced by register spilling. We add the spill temporaries and their access sequences to the Access Graph of the first pass.

Phase four—*Offset Assignment*—assigns stack offsets to spilt locals and temporaries, guided by the access paths generated in the previous phase.

Finally, *Register Assignment* maps the virtual registers onto the physical registers of the target architecture. Again, we use the access paths to guide the assignment of registers to the MMA instructions, since most target MMA instructions enforce some order on the register list.

The output of this chain of phases is a CFG from which target code can be emitted. Where the register assigner has been unable to comply with the constraints of the given target’s MMA instruction register ordering, we decompose a single provisional MMA instruction back into a number of smaller MMA instructions, or even single loads or stores.

3.7 Complexity of Heuristic Algorithm

SOLVEMMA processes each edge in $E \subseteq V \times V$, which is potentially quadratic in the number of variables in the program. Thus we require SOLVEMMA to be an efficient algorithm else it will be too costly for all but trivial programs. Here, we show that SOLVEMMA has similar complexity to SOLVESOA.

Lemma 1. *The running time of SOLVEMMA is $O(|E| \log |E| + |L|)$, where E is the number of edges in the Access Graph, and L is the number of variable accesses in the given program.*

⁶ We assume that local user variables have been mapped to virtual registers, except in the case of address-taken variables, which are turned into one-element arrays.

Proof. SOLVEMMA is derived from SOLVESSA, whose complexity has been shown to be $O(|E| \log |E| + |L|)$. Thus we only consider the additional complexity introduced in the construction of the MMA Access Graph and in lines 10–11 and 13–16 of Figure 2. For every variable access $l \in L$ there can be at most 1 adjacent variable access. Lines 10, 11 and 14 incur a constant cost per edge, as does line 13 in a well-implemented program. Lines 15–16 together walk paths to convert them into directed paths. A path can be converted from undirected to directed at most once. The total complexity is then $O(|E| \log |E| + |E| + |E| + |E| + |L|)$, *i.e.*, $O(|E| \log |E| + |L|)$. \square

Using path elements [13] the test for whether an edge e forms a cycle in C is reduced to testing whether the head and tail vertices of e are common to a single path element. This test can be performed in constant time.

4 Multiple Memory Access on the VSDG

The previous section applied MMA optimization to a program in CFG form. The transformations that are possible are constrained by the precise ordering specified by the CFG. In this section we apply MMA optimization to the Value State Dependence Graph, which has fewer constraints on the ordering of instructions.

4.1 The Value State Dependence Graph

The Value State Dependence Graph (VSDG) [8] $G = (N, E_V, E_S, \ell, N_0, N_\infty)$ is a directed graph of a single procedure (as for the CFG) consisting of operation, loop (θ) and merge (γ) nodes N , special entry (N_∞) and exit (N_0) nodes, node labelling function ℓ , and value (E_V)- and state (E_S)-dependency edges.

Value dependency edges indicate the flow of values between nodes, and must be preserved during register allocation and code motion.

State dependency edges represent the *essential* sequential dependency required by the original program. These edges both fix an issue in the Value Dependence Graph (VDG) [17], and are useful for other purposes (*e.g.*, our Combined Register Allocation and Code Motion (RACM) algorithm [8]).

The VSDG inherits from the VDG the property that a program is implicitly represented in Static Single Assignment (SSA) form [5]: a given operator node, n , will have zero or more value-dependent-successors using its value.

Figure 4 shows the VSDG of the example program from Figure 1, showing memory load, memory store and add nodes, with value- and state-dependency edges, together with the resulting Access Graph and memory layout.

4.2 Modifying SOLVEMMA for the VSDG

The VSDG under-specifies the ordering of instructions. Thus we can reformulate the four criteria of Definition 3.2 to “(1) $i \in \alpha(p)$ and $j \in \alpha(q)$ are in the same

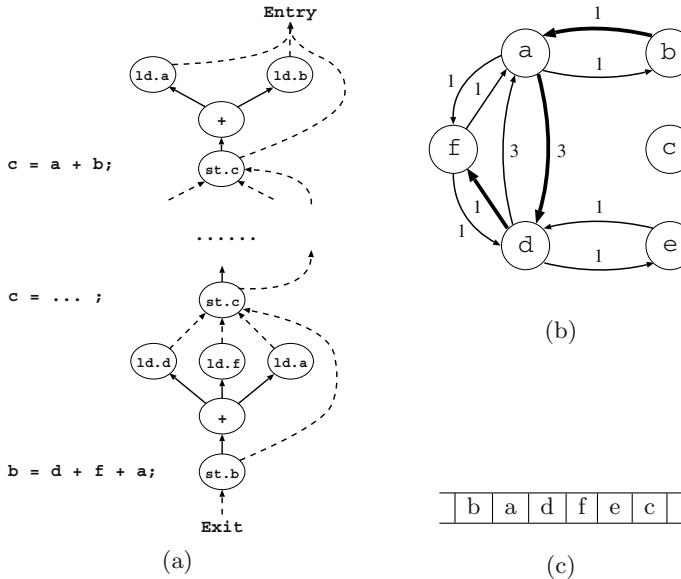


Fig. 4. VSDG of Figure 1. The solid lines in (a) are value dependencies, and the dashed lines state dependencies (the middle portion of the graph is not shown, but can be readily constructed). The labels of the memory load ‘`ld`’ and memory store ‘`st`’ nodes include the variable name after the period. The Access Graph of (a) is shown in (b), with the covered edges in bold, resulting in memory layout (c).

γ - and θ -dominated regions⁷; (2) $op(i) = op(j)$; (3) for loads, j has the same state-dependent node as i ; for stores, j state-depends on i ; and (4) j is not value-dependent on i .

For example, in “`x = v[a + b]`” there are three loads—one each for `a`, `b` and the indexed access into array `v[]`. But the order of the first two loads is unspecified, and so either can precede the other. This is represented in the Access Graph by two edges (a, b) and (b, a) of equal weight.

We gain two benefits from using the VSDG. The first is due to the state dependency edges defining the necessary ordering of loads and stores. For example, if a group of loads all depend on the same state, then those loads can be scheduled in any order; the VSDG *underspecifies* the order of the loads, allowing MMA optimization to find an order that benefits code size.

The second benefit is due to the separation of value dependency and state dependency. Such a separation facilitates a simple method of code motion by inserting additional serializing edges into the VSDG. For example, we can hoist an expression from between two stores, allowing them to be combined into a single MMA store.

The VSDG does not change the SOLVEMMA algorithm itself, but we argue that it is a better data structure, and allows greater flexibility in the mixing of

⁷ These are equivalent to basic blocks in the VSDG.

phases within the compiler. For example, the register allocation phase [8] can favour an instruction ordering which allows provisional MMA instructions to become actual MMA instructions.

5 Target-Specific MMA Instructions

The generic MMA instructions (§3.1) do not specify any addressing mode, since the underlying algorithm is neutral in this respect. Its purpose is to combine multiple instructions into a single instruction to reduce code size.

Tailoring the algorithm for a given architecture requires adding constraints to the code generator and register assigner. For example, the PowerPC’s `1mw` and `stmw` instructions require a contiguous block of registers ending in `R31`.

The ARM Thumb LDM and STM instructions use a post-incremented register. Thus we only consider access paths where the address is increasing. `Raddr` is updated with the address of the word *after* the last word accessed by the MMA instruction. This gives another form of register re-use similar to GOA which we resolve opportunistically: after an MMA instruction the base register may be available for re-use later, saving an address computation.

6 Experimental Results

The SOLVEMMA algorithm has been implemented as a transformation tool within our VSDG tool chain. The tool chain consists of an ANSI C compiler (based on LCC [7]), a multi-pass optimizer, an ARM Thumb code generator, and a combined register allocator and code motion pass (an implementation of our earlier RACM algorithm [8]).

We chose the ARM Thumb because its MMA instructions (`LDMIA` and `STMIA`) are a subset of those in the full ARM instruction set, so any implementation will be applicable to both instruction sets, and because there is greater interest in producing compact code for the Thumb than for the ARM.

6.1 Motivating Example

Our first experiment is shown in Figure 5. It is a function which performs some operation on a global array, as might be found in network packet processing. SOLVEMMA combined all four loads into a provisional LDM instruction, and similarly for the four stores. However, the code generator found it cheaper (avoiding `mov` instructions) to undo the provisional LDM and emit four separate load instructions. Using classical optimizations not yet implemented in our experimental compiler we can remove the four instructions highlighted in Figure 5(b) by modifying the final `add` instruction.

```

void bufswap(int * array) {
    int i;
    for ( i = 0; i < 256; i += 4 ) {
        int t1, t2, t3, t4;
        t1 = array[i];
        t2 = array[i+1];
        t3 = array[i+2];
        t4 = array[i+3];
        array[i] = t4;
        array[i+1] = t3;
        array[i+2] = t2;
        array[i+3] = t1;
    }
}

```

(a) C source

```

bufswap PROC
    push    {r4-r6,LR}
    mov     r1, #0
    tn_2   cmp    r1, #255      **
    bgt    tn_1      **
    lsl     r2, r1, #2      **
    add     r6, r0, r2      **
    ldr     r5, [r6, #0]
    ldr     r4, [r6, #4]
    ldr     r3, [r6, #8]
    ldr     r2, [r6, #12]
    stmia  r6!, {r2-r5}
    add     r1, r1, #4
    b      tn_2
tn_1   pop    {r4-r6,PC}

```

(b) Thumb code (14 instructions)

Fig. 5. An illustrative example of MMA optimization of the function in (a). The output of our compiler (b) shows that we combine the four stores into a single STMIA instruction. This code compares favourably with that produced by the ARM commercial compiler, which required 17 instructions and one extra register; using classical optimizations our code gives 10 instructions.

6.2 Discussion of Benchmark Cases

The SOLVEMMA algorithm performs best on spill-code and global-variable-intensive code, where a significant proportion of the instructions can be combined into MMA instructions. Of note is that SOLVEMMA degrades gracefully—in the worst case a provisional MMA instruction is decomposed into a corresponding number of load or store instructions.

We applied our compiler to a number of test cases from MediaBench [11], which provide opportunities of our algorithm to reduce code size. The results are shown in Table 1 for individual functions where there was a reduction in code size. In all other cases, while SOLVEMMA identified many provisional MMA instructions, the limitations of the ARM Thumb addressing modes generally need to combine three instructions to save one instruction. At worst, the code generator emits a group of single loads or stores which, while not reducing code size, neither does it increase.

In the first case, `Gsm_Coder()`, the saving was through the use of a multiple push (a special form of STM using the stack pointer as address register with pre-decrement) to place two function arguments onto the stack prior to a function call.

The remaining cases utilise LDM or STM instructions to reduce code size. Further examination of the generated ARM code indicates places where provisional MMA instructions have been decomposed into single instructions during register allocation.

Table 1. Measured behaviour of MMA optimization on benchmark functions. Numbers indicate instruction counts; both ARM and GCC compilers were run with space optimization selected.

Source	Without MMA	With MMA	ARM	GCC
<code>code.c::Gsm_Coder()</code>	93	92	91	105
<code>decode.c::Gsm_Decoder()</code>	86	81	66	82
<code>gsm_encode.c::gsm_encode()</code>	561	559	430	552
<code>lpc.c::Reflection_coeffs()</code>	193	191	211	234

We also compared our compiler to both GCC and the ARM Ltd. commercial Thumb compiler on the same benchmarks. We achieve almost 13% *smaller* code than the GCC compiler, by spilling fewer callee-save registers and using MMA optimization. In contrast, we produce around 9% more instructions than the ARM compiler, which uses peephole optimization, a carefully designed register allocator, narrow sub-word optimizations, and other optimizations which are not the focus of this work.

7 Conclusions and Further Work

This paper introduces the SOLVEMMA algorithm as a tool for combining several memory access instructions into a single MMA instruction. Using a technique similar to that of Liao’s SOLVESOA algorithm, we both identify loads or stores that can be combined into single MMA instructions and guide stack frame layout. Implemented as a transformation within our VSDG tool chain targeting the ARM Thumb processor, we achieve up to 6% code size reduction, and never increase code size.

One question that remains unanswered is which of the currently available MMA instructions offers the best results for code compaction. In our choice of the ARM Thumb we have a single address register and a bitmap to specify the desired data registers. In contrast the MMA instructions of the PowerPC support register+displacement addressing and specify the start index of a block of contiguous registers ending in R31.

The current implementation of SOLVEMMA takes a simple approach to alias analysis, considering only loads or stores where we can directly infer their address relationship from the VSDG. More aggressive alias analysis should identify yet more opportunities for combining loads and stores into MMA instructions.

Acknowledgements. The research was supported with a grant from ARM Ltd. Thanks are due to Lee Smith for reminding us of the importance of the problem, to the Cambridge Programming Research Group for helpful discussions on the SOLVEMMA algorithm, to Kate Taylor for reviewing an early draft of this paper, and to the CC’04 reviewers for their helpful remarks.

References

1. *i486 Processor Programmer's Reference Manual*. Intel Corp./Osborne McGraw-Hill, San Francisco, CA, 1990.
2. AHO, A., HOPCROFT, J., AND ULLMAN, J. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.
3. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
4. BARTLEY, D. H. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software—Practice and Experience* 22, 2 (February 1992), 101–110.
5. CYTRON, R. K., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently Computing the Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Programming Languages and Systems* 12, 4 (October 1991), 451–490.
6. FISHER, R. J., AND DIETZ, H. G. Compiling For SIMD Within A Register. In *Proc. 11th Annual Workshop on Languages and Compilers for Parallel Computing (LCP'98)* (August 1998), vol. 1656 of *LNCS (Springer-Verlag)*, pp. 290–304.
7. FRASER, C. W., AND HANSON, D. R. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.
8. JOHNSON, N., AND MYCROFT, A. Combined Code Motion and Register Allocation using the Value State Dependence Graph. In *Proc. 12th International Conference on Compiler Construction (CC'03)* (April 2003), vol. 2622 of *LNCS (Springer-Verlag)*, pp. 1–16.
9. KISSELL, K. MIPS16: High-density MIPS for the Embedded Market. In *Proc. Conf. Real Time Systems (RTS'97)* (1997).
10. KOSEKI, A., KOMATSU, H., AND NAKATANI, T. Preference-Directed Graph Coloring. In *Proc. ACM SIGPLAN 2002 Conference on Prog. Lang. Design and Implementation* (June 2002), ACM Press, pp. 33–44.
11. LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. 30th Intl. Symp. Microarchitectures (MICRO'97)* (December 1997), ACM/IEEE, pp. 330–335.
12. LEUPERS, R., AND MARWEDEL, P. Algorithms for Address Assignment in DSP Code Generation. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design* (1996), IEEE Computer Society Press, pp. 109–112.
13. LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. Storage Assignment to Decrease Code Size. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation* (June 1995), ACM Press, pp. 186–195.
14. RAO, A., AND PANDE, S. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation* (1999), ACM Press, pp. 128–138.
15. SEAL, D. *ARM Architecture Reference Manual*. Addison-Wesley, UK, 2000.
16. UDAYANARAYANAN, S., AND CHAKRABARTI, C. Address Code Generation for Digital Signal Processors. In *Proceedings of the 38th Conference on Design Automation* (2001), ACM Press, pp. 353–358.
17. WEISE, D., CREW, R. F., ERNST, M., AND STEENSGAARD, B. Value Dependence Graphs: Representation Without Taxation. In *ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Langs (POPL)* (January 1994), ACM.

Integrating the Soot Compiler Infrastructure into an IDE^{*}

Jennifer Lhoták, Ondřej Lhoták, and Laurie Hendren

Sable Research Group, McGill University, Montreal, Canada
`{jlhotak, olhotak, hendren}@sable.mcgill.ca`

Abstract. This paper presents the integration of Soot, a byte-code analysis and transformation framework, with an integrated development environment (IDE), Eclipse. Such an integrated toolkit is useful for both the compiler developer, to aid in understanding and debugging new analyses, and also for the end-user of the IDE, to aid in program understanding by exposing semantic information gathered by the advanced compiler analyses. The paper discusses these advantages and provides concrete examples of its usefulness. There are several major challenges to overcome in developing the integrated toolkit, and the paper discusses three major challenges and the solutions to those challenges. An overview of Soot and the integrated toolkit is given, followed by a more detailed discussion of the fundamental components. The paper concludes with several illustrative examples of using the integrated toolkit along with a discussion of future plans and research.

1 Introduction

In this paper we present the integration of Soot [23], a byte-code analysis and transformation framework, into an integrated development environment (IDE), Eclipse. Traditionally, optimizing compilers and optimizing compiler toolkits concentrate on providing the compiler developer with relevant intermediate representations (IRs) and support for a variety of program analyses and transformations. Although this support is key for the development of new compiler analyses, it does not provide the compiler writer with tools for easily visualizing the results of their analysis (which is very useful for debugging), nor does it provide a simple way of conveying the analysis information to an end-user.

On the other hand, IDEs often produce various visual representations of the code to help the developer understand it. However, IDEs do not usually support the low-level intermediate representations and advanced compiler analyses found in optimizing compilers. Thus most visual representations in IDEs are based only on structural analysis of a high-level representation of the code, so they cannot communicate to the developer the wealth of semantic information that is deduced by an optimizing compiler.

By integrating a compiler toolkit with an IDE, we can provide a much richer environment for both the compiler developer and for the end-user. For the compiler writer, we can provide mechanisms for easily displaying and browsing the results of compiler analyses. This is very useful for both students who are developing their first simple analyses and for more advanced researchers and developers who are developing more

* This work is funded, in part, by an IBM-sponsored Eclipse Innovation Grant, and by NSERC.

complex analyses. For the end-user, we can provide interesting ways of conveying program analysis information back to them. For ordinary users we can provide information to help them understand the behaviour or potential problems in their program, based on analyses that are more complex than currently supported in IDEs. For example, we could display information about potential array bounds problems. For advanced users we can provide more detailed information about the results of program analyses. For example, we can expose side-effect or alias information.

1.1 Challenges

There are three major challenges to overcome in order to make a useful integration of a compiler toolkit and an IDE. First, the compiler developer would like to concentrate mostly on developing and implementing an analysis or transformation on a convenient IR and should not be burdened with the complexities of how this information will be displayed in the IDE.

Second, the compiler analysis/transformation will likely be performed on a lower-level IR than the Java source and for debugging purposes the analysis results should be displayed on this IR. However, the end-user is not likely to understand the low-level IR and will likely want to view information at the Java source level. Thus, there must be a simple and transparent mechanism for keeping the correspondence between the lower-level IR and the Java source files, and for displaying the analysis information at both levels.

Third, the compiler framework and the IDE are separate pieces of software and the interface to make them interact cleanly must be carefully designed. In particular, we need simple mechanisms for conveying information from the IDE to the compiler framework, and analysis information from the compiler framework back to the IDE.

1.2 Contributions

In this paper we present our integration of the Soot compiler framework into the Eclipse IDE and we show how we have addressed the three major challenges given above. Although we discuss our approach with respect to a specific IDE, namely Eclipse, the general strategy should apply to other IDEs as well.

We tackle the challenge of isolating the compiler developer from complex IDE display concerns by providing three new kinds of abstract tags which can be attached to classes, fields, methods, statements or values in the low-level IR. The compiler developer only has to worry about tagging the IR with the results of a compiler analysis (a simple process) and the mechanism for conveying the information to the IDE and displaying the information in the IDE is handled automatically by the integrated toolkit.

For the challenge of supporting the display of analysis information for both the low-level IR and the source code, we provide automated support for transparently keeping the correspondence between the low-level IR statements and the correct position within the source code. This is done by providing a new front-end based on Polyglot which parses Java source and produces the lower-level Jimple IR with tags already added to give the line number and column offset in the Java Source.

Finally, we address the challenge of combining the Soot framework and the Eclipse IDE by providing integration code comprised of three main modules (launcher, IR editor and attribute handler), and by defining an XML specification for both the command-line options to Soot and for the attribute information which conveys line numbers, column offsets and analysis information.

1.3 Paper Roadmap

The remainder of the paper is structured as follows. In Section 2, we provide an overview of the structure of Soot and the structure of the integration code. In Section 3, we discuss our three new kinds of tags. Section 4 discusses the mechanisms we have developed for communicating between the IDE and the compiler framework. In Section 5, we provide four illustrative examples of using the integrated toolkit. Section 6 gives an overview of related work and Section 7 provides conclusions and future work.

2 Structure of Soot and the IDE Plugin

2.1 Soot

Soot [23] was originally developed as a framework to allow researchers to experiment with analyses and optimizations of Java bytecode, and to provide a common infrastructure to compare results. It was later extended with an annotation framework, which allows analysis results to be encoded as class file annotations for use by other tools [17]. Many standard and advanced analyses and optimizations have been implemented both at McGill and at many other institutions. Examples from McGill include array bounds analysis [18], variable type analysis [21] and pointer and side-effect analysis [13]. In order to make Soot useful in the IDE integration, and to support visualization of program analysis results, several modifications and extensions were required. Figure 1 shows a high-level view of Soot, including the new parts that have been developed to support the IDE integration and visualizations (new parts are shaded in dark grey or red). As indicated at the top of Figure 1, Soot takes as input the program to analyze in one of three forms (Java class files, Jimple files or Java source files) and the command-line arguments which specify which input files to process, what phases and options to enable in Soot, and what kind of output to create.

Inside Soot, the input program is converted into the Jimple IR, which is a three-address typed intermediate representation that is convenient for program analysis. After Jimple has been created, various analyses and transformations can be applied to the program, resulting in Jimple code that is associated with analysis information. Soot supports a variety of intraprocedural analyses, as well as interprocedural analyses such as points-to and side-effect analysis. Compiler developers can add new analyses and/or transformations to this phase quite easily.

The next phase in Soot attaches tags to the Jimple IR. The original purpose of tags was as a mechanism for conveying flow information, such as array bounds checking information, to a VM. However, for the purposes of the IDE integration, we have developed three new kinds of tags which are used to convey program analysis information to

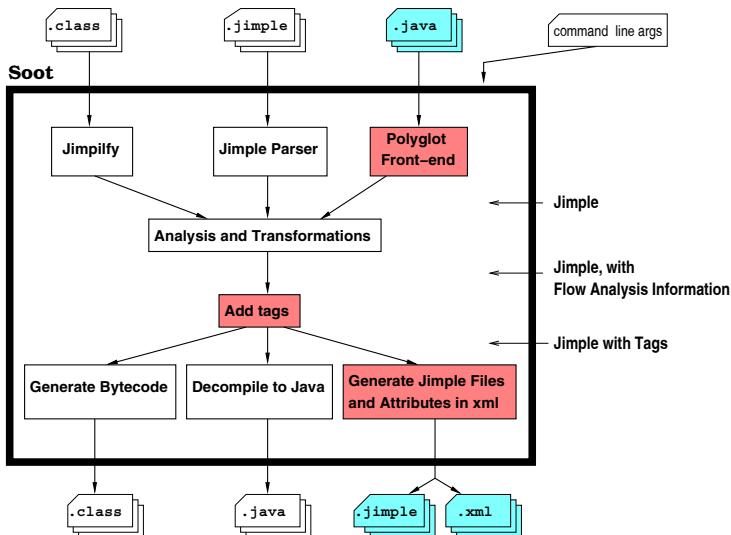


Fig. 1. Overview of Soot

the IDE. We have also added a collection of built-in tagging modules based on analyses already available in Soot. In addition, it is a very simple process for the compiler writer to create appropriate tags to encode the results of new analyses he/she has added to Soot. These new visualization tags are discussed in detail in Section 3.

As indicated at the bottom of Figure 1, Soot can produce three kinds of outputs: class files, Java source files or Jimple files. In addition to these output files, Soot communicates the analysis results from the tags to the Eclipse plugin using XML files. The details of this communication are described in Section 4.

2.2 Soot Plugin for Eclipse

The Soot framework was originally designed as an ordinary Java application that was run from the command-line. In order to keep this functionality, as well as provide integration with an IDE, we created a Soot Plugin for Eclipse.¹ As illustrated in Figure 2, the plugin is composed of two main pieces: Soot (which can serve as a stand-alone application when used by itself), as outlined in Figure 1, and the Soot Interface which provides the glue between the Eclipse platform and Soot. The *Soot Interface* itself is composed of three main parts: the *Soot Launcher*, the *IR Editor* and the *Attribute Handler*.

The *Soot Launcher* module handles calling Soot with the appropriate command-line arguments, interfaces with the Eclipse resource manager to determine the location of the input files and libraries, determines the correct location for the output files, and handles all messages sent to standard output by Soot.

The command line version of Soot provides many options to specify its behaviour. Users of the Eclipse plugin can set these same options in two different ways. First, we

¹ The Eclipse plugin is distributed as part of Soot, which is publicly available under the LGPL at <http://www.sable.mcgill.ca/soot/>.

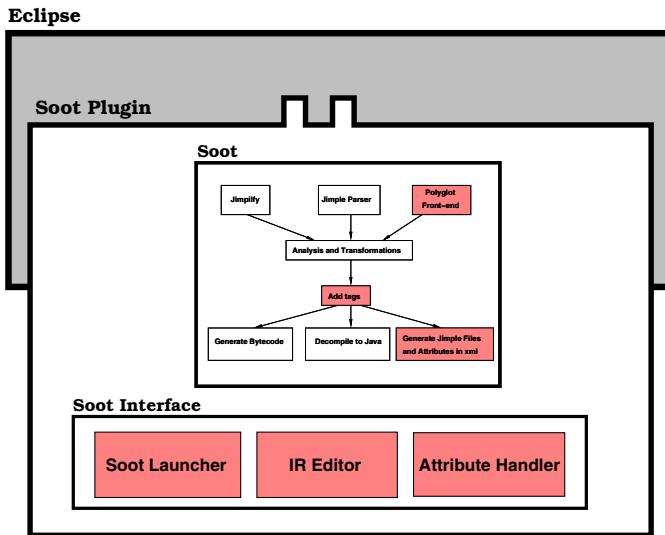


Fig. 2. Soot as an Eclipse Plugin

have made several common settings (such as generating Jimple output and decompiling) available as menu items. For more advanced users, the plugin includes a dialog containing all the Soot options. The settings in this dialog can be saved as named configurations, to make it easy to run Soot many times with the same options. The configurations are persistent, so they are available in subsequent Eclipse sessions.

Soot has a large number of options, and they often change as analyses and transformations are added to Soot. An important engineering challenge was how to maintain consistency between Soot and the plugin. Our solution was to create an XML file specifying and documenting all the options. This file is used to **automatically** generate the option handling code for both Soot and the Eclipse plugin, as well as documentation, both printed and on-line (including tooltips in the options dialog). Using a single source for the option information ensures that Soot, the plugin, and the documentation remain consistent.

The *IR Editor* handles displaying the several different IR's in Soot, providing syntax highlighting and a content outliner. The outliner is useful as the IR's are often much longer than the original source, presenting the need to easily navigate to particular methods. The IR editor also displays the results of analyses in visual ways.

The *Attribute Handler* is responsible for reading analysis results from Soot, organizing them, and displaying them in a visual way. When triggered, it reads an XML file which Soot has produced containing all the analysis results. The information must then be organized according to which parts of the code it applies to, so that it can be displayed in the appropriate places. The *Attribute Handler* then displays the analysis information in the code using popup tooltips, colour highlighting, popup link menus, and icons.

Figure 3 provides a screenshot showing a typical use of the integrated toolkit. In the foreground (upper left), one can see the Soot dialog box which is used for setting

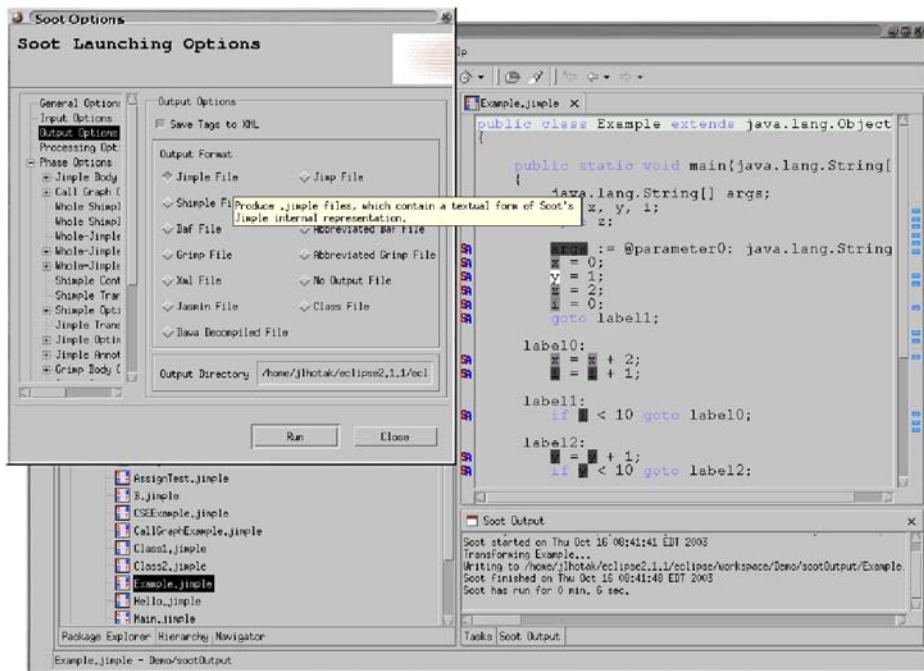


Fig. 3. Soot-Eclipse Screenshot

the options for running Soot. In this case, the user is setting the *Output Options*. As the user moves the cursor over the various output options, a tooltip is displayed which documents the option. In this case, the mouse is over the option for *Jimple File*, and so the appropriate descriptive tooltip is given.

In the background, one can see the typical Eclipse display. On the left side is the Package Explorer listing the files in the current project, including some .jimple files generated by Soot. On the right side is the Intermediate Representations Editor with syntax highlighting editing a Jimple file, which was produced by running Soot with a parity analysis enabled. As described further in Section 5.1, the compiler developer simply tags the variables in the IR with Colour Tags based on the results from the analysis, and the integrated toolkit highlights the variables in the code with different colours representing results of the analysis. The icons in the left margin labelled SA indicate lines of code for which analysis information is available. Finally, the Soot Output view appears in the bottom right corner, showing the output of the last execution of Soot.

3 Visualization Primitives

One of the key requirements in the design of Soot's visualization framework was that it be general enough to display the results of many kinds of analyses, including ones that we have not yet thought of. This is important for several reasons. First, Soot is

not only a compiler, but rather a compiler *framework*. While it includes many standard analyses, it is mainly valuable as a tool to help compiler researchers easily implement their own analyses. Therefore, the visualization framework that we design today must be adaptable to analyses that will be designed tomorrow by other researchers. Second, one of the most important uses of Soot is in teaching program analysis, both at McGill and at other institutions, and the visualization framework is one way in which we are improving Soot for teaching. In the compiler course at McGill, one assignment requires students to implement a dataflow analysis of their choice within Soot. The students should be able to display the results of their analysis as they are developing it using the visualization framework. Therefore, the framework must be able to support the arbitrary analyses that the students may choose to implement. To make the visualization framework general with respect to different analyses, we identified three *visualization primitives* to convey the different kinds of information likely to be generated by analyses. The primitives are intended to be general enough so that any analysis information can be presented with them. In Section 5, we describe several example analyses whose results are displayed with these primitives. Should we encounter an analysis for which these three primitives are insufficient, we would consider adding a new primitive, but we would again look for a general primitive that could be used to display the results of many different analyses.

For each visualization primitive, Soot includes a type of abstract tag, which is used to store the information to be displayed. These tags can be attached to arbitrary classes, methods, fields, statements, and values in any of the Soot intermediate representations. The analysis developer needs only to choose the visualization primitive, create a tag containing the information to be displayed, and attach it to the appropriate object in the intermediate representation.² Soot's annotation framework propagates the tags between the intermediate representations. The visualization framework collects the tags, and displays them in the appropriate way depending on which visualization primitive they represent.

3.1 String Tags

The simplest way to present analysis information to the user is using an arbitrary string of text. The String Tag visualization primitive allows arbitrary text to be associated with an object in the intermediate representation. In previous versions of Soot, the only way to output analysis results was to print them as comments in the intermediate representation, often leading to a very cluttered output. Within Eclipse, however, this text is not displayed as part of the intermediate representation itself, but as popup tooltips that appear when the mouse is over the corresponding piece of code. The user can trace through the analysis results by moving through the code with the mouse. An icon in the margin indicates to the user which lines of code contain analysis information.

Almost any analysis information can be encoded using a String Tag, since almost any information can be described using text. Even when displaying information that is more conveniently displayed using one of the other tags, we encourage analysis developers to include String Tags explaining the analysis results to supplement these other tags.

² A complete example of the code for attaching tags is given in Section 5.1.

A String Tag is very easy for an analysis developer to create. Its constructor requires only the string to be displayed. It is attached to a class, method, field, statement or value with a single call to its `addTag()` method. For example to tag a method `m` as not reachable, one could use `m.addTag(new StringTag("Method not reachable"));`; or assuming some flow analysis information has been computed and stored in a map called `analysisInfo`, one could attach a textual representation of the flow analysis information to a statement `s`, using `s.addTag(new StringTag(analysisInfo.getFlowBefore(s).toString()));`.

3.2 Colour Tags

Many analyses assign one of a small collection of possible values to each small section of the intermediate representation. For example, a live variable analysis assigns either *live* or *dead* to each local variable at each program point. This type of information can be conveniently displayed by highlighting the relevant section of the intermediate representation with a colour representing the value. For example, each appearance of a variable in the code could be coloured red if it is dead after the statement in which it appears, and green if it is live.

To highlight a class, method, field, statement or value, the analysis developer needs only to select a colour (either from a collection of pre-defined colours, or by specifying an arbitrary triple of red, green, and blue values), create a Colour Tag of that colour, and attach it with a call to `addTag()`. For example, to colour an unreachable method `m` red, one could use `m.addTag(new ColorTag(ColorTag.RED));`.

3.3 Link Tags

Some analyses produce results which take the form of references to other portions of the code. For example, a reaching definitions analysis produces, for each use of a variable, the set of definitions reaching the use, where each definition is some statement in the code defining the variable. This type of information is most naturally conveyed using hyperlinks to the code being referenced. To allow the results of an analysis to be presented as hyperlinks, the visualization framework includes a Link Tag primitive. Each Link Tag requires a textual description and a target statement, method, or field (which may be in the same or a different class); it can be attached to a statement, method, or field. For example, to create a link between a use of a variable and its defining statement, one could use `useStmt.addTag(new LinkTag("Definition of "+variable, defStmt, className));`.

The same value, statement, field or method can be tagged with multiple Link Tags. For example, a statement may have multiple reaching definitions. The textual descriptions of all Link Tags appearing at a given line are shown in a popup tooltip when the mouse moves over the line. In order to select one of the links, the user clicks on the SA icon in the margin to produce a popup menu of all the Link Tags with their textual descriptions. By clicking on a specific item in this menu, the user can follow one of the links; the cursor is moved to the target of the link. Eclipse also provides forward and back navigation buttons like a web browser. As shown later in our example in Section 5.2, this functionality is very useful for navigating call graphs.

4 Communicating Analysis Information from Compiler to IDE

A compiler or compiler framework like Soot performs analyses and stores analysis results on a low-level IR such as Jimple. However, the IDE does not have direct access to this internal intermediate representation; it deals with a textual representation of the program source, and perhaps a textual representation of the intermediate representation. In this section, we describe how the analysis information is mapped from the internal representation of the compiler to the textual representations managed by the IDE. In earlier work [17], Soot was extended with a framework for encoding analysis results in its intermediate representation by attaching tags to objects of the representation. This framework propagates the tags between the various intermediate representations inside Soot, and originally, it was used to encode the analysis information in the class files produced as output.

For the Eclipse visualization framework, we were able to leverage this annotation framework. The key component that had to be added was a method to encode the tags containing analysis information in such a way that Eclipse could read them, and match them to the appropriate positions in the textual version of the intermediate representation. To achieve this, we modified Soot’s intermediate representation output code to output, along with the text of the intermediate representation itself, an XML file containing information from all the tags attached to the intermediate representation. The output code encodes the line and column positions in the text of each object of the intermediate representation as well as encoding the appropriate position for each tag. The Soot interface to Eclipse then needs only to parse the XML file to determine which tags should be displayed at which line and column position of the intermediate representation.

Mapping analysis information back to the original source code is a more difficult problem, compounded by the fact that traditionally, the input to Soot has been pre-compiled Java bytecode, rather than the original source code. One way to do this is to make use of information stored in Java bytecode attributes. The Java Virtual Machine Specification [14] defines the `LineNumberTable` attribute, which maps each bytecode instruction to the line of source code from which it was produced. Soot parses this attribute and encodes the mapping as a source line number tag attached to each IR instruction. Since these tags are propagated through the various intermediate representations, they are still present when Soot outputs analysis information into the XML file. At this point, the statements of the intermediate representation have two types of tags attached to them: tags containing the analysis information, and tags containing source line number information. When Soot records the analysis information in the XML file, it can add the source line number information with each tag. This makes it possible for the Eclipse plugin to display the tag not only in the correct position in the intermediate representation, but also with the correct line of the source code.

String Tags and Link Tags tend to be coarse-grained enough that it is usually sufficient to display them with the correct line of the source code. However, Colour Tags are often used to highlight a small part of a line, such as a single variable. Unfortunately, the Java class files do not preserve such detailed information about the positions of variable names in the source code. In order to have sufficient information to display Colour Tags correctly in source, Soot must accept the original source code as its input, rather than bytecode. We therefore constructed a Java source front-end for Soot, which

converts Java source code directly to the Jimple intermediate representation. The front-end is based on Polyglot [16], a semantic analyzer for Java source, to which we added a Jimple code generator back-end. For consistency with the rest of Soot, this Java-to-Jimple compiler encodes the source line and column information for each object of Jimple using Soot tags. This means that Soot's annotation framework automatically propagates the information between intermediate representations, and it is available when the output code is generating the XML file containing analysis information. The source line and column information is encoded in the XML file along with the intermediate representation line and column information, so the Eclipse plugin can display the analysis information in the correct place in both the intermediate representation and the source code.

5 Example Applications of the Integrated Framework

In this section, we present several examples of applications of the integrated framework. The applications illustrate typical uses for teaching (Section 5.1), program understanding (Sections 5.2 and 5.3) and more advanced research analysis (Section 5.4).

5.1 Parity Analysis

In the optimizing compiler course at McGill, students are asked to implement a dataflow analysis of their choice in Soot. We also encourage the students to use the Eclipse plugin to visualize the results of their analysis.

Parity analysis is one example of an analysis that students have chosen to implement. For each integer expression appearing in a method, the analysis computes whether its value is always even, always odd, or could be either (top). Typically the amount of code needed to visualize an analysis is very small and it is easy to write. The code needed to visualize the results of parity analysis is given in Figure 4. The code iterates through the statements in the method, and for each statement, it iterates through the values defined and used. Each use is tagged with the analysis result before the statement, and each definition is tagged with the analysis result after the statement.³ The addTags() method actually encodes the results in tags. First, a String Tag is added for each value indicating its parity in textual form. These String Tags will be automatically displayed in Eclipse as tooltips. Then, depending on the parity, a colour is chosen and a Colour Tag is added. These Colour Tags will cause the variables to be coloured appropriately. Figure 3 showed the Jimple code with the Colour Tags indicating parity information.

In general, we find that students find it easy to add the tagging information and the resulting Eclipse displays aids in their understanding and debugging of the analysis, as well as making the exercise more fun and rewarding.

³ For example, in a statement of the form $x = x + y$, the parity of the uses of x and y are stored in flow information before the statement, whereas the parity of the definition of x is stored in the flow information after the statement.

```

protected void internalTransform( Body b, String phaseName, Map options)
{ ParityAnalysis pa = new ParityAnalysis(new BriefUnitGraph( b )); // compute analysis
  for( Iterator it = b.getUnits(); it.hasNext(); ) { // iterate over all stmts
    Stmt stmt = (Stmt) it.next();
    // Tag all definitions with flow information after statement.
    addTags( stmt.getDefBoxes(), (Map) pa.getFlowAfter( stmt ) );
    // Tag all uses with flow information before statement.
    addTags( stmt.getUseBoxes(), (Map) pa.getFlowBefore( stmt ) );
  }
}

private void addTags( Collection boxes, Map parityMap ) {
  for( Iterator it = boxes.iterator(); it.hasNext(); ) { // iterate over all values
    ValueBox box = (ValueBox) it.next();
    Value value = box.getValue();
    // get the flow information
    String parity = (String) parityMap.get( value );
    if( parity == null ) return; // No parity information for this value.
                                // (only computed for variables of int type)
    // add a String Tag
    box.addTag( new StringTag( value.toString() + " is " + parity + "." ) );
    // add a Color Tag
    if( parity.equals( ParityAnalysis.EVEN ) )
      box.addTag( new ColorTag( ColorTag.YELLOW ) );
    else if( parity.equals( ParityAnalysis.ODD ) )
      box.addTag( new ColorTag( ColorTag.BLUE ) );
    else if( parity.equals( ParityAnalysis.TOP ) )
      box.addTag( new ColorTag( ColorTag.RED ) );
    else if( parity.equals( ParityAnalysis.BOTTOM ) )
      box.addTag( new ColorTag( ColorTag.GREEN ) );
    else throw new RuntimeException( "Unknown parity value " + parity + "." );
  }
}

```

Fig. 4. Code to visualize parity analysis results

5.2 Call Graph and Parameter Alias Analysis

Soot includes a whole-program analysis framework for constructing call graphs and performing points-to analysis [13, 21]. The information computed by this framework can be particularly useful to help developers understand their code and find bugs in it. In this section we demonstrate how the call graph and parameter alias visualizations were used to find a bug in one of our benchmark programs, `telecom`. The `telecom` program simulates phone calls taking place in a public telephone network, and it is an extended, multi-threaded version of a benchmark distributed with AspectJ [1] that was submitted as an example benchmark for an optimizing compilers course.

The call graph visualization presents the call graph computed by Soot using Link Tags. This allows the developer to see the possible targets of call sites, including virtual call sites, and easily navigate the call graph by following the links. Reverse links are also

included from each method to all the call sites from which it may be called. Although most IDEs provide search features to help developers find specific methods, these are often based only on a textual search, ignoring the semantics of the class hierarchy, or, at best, are based on an imprecise call graph constructed using Class Hierarchy Analysis [9]. On the other hand, the call graph constructed by Soot can be made much more precise, using techniques such as Rapid Type Analysis [2], Variable Type Analysis [21], or even more precise points-to analyses.

Soot's points-to framework [13] can be used to compute may-alias information for pointers. The parameter alias visualization presents a subset of this information. Specifically, for each method with pointers as parameters, it indicates, using Colour Tags, which pointer parameters may point to the same memory location when the method is invoked. Parameters which may point to the same location are coloured with the same colour, while parameters known to point to distinct locations are coloured with different colours.

Figure 5(a) shows the Call Graph and Parameter Alias visualizations applied to the `Call()` constructor from the telecom example. Notice that the two parameters, `caller` and `receiver` are both the same colour, indicating that they may be aliased.⁴ This appears to be a possible error; in a telephone system, it should not be possible for a caller to call himself. In order to determine if there really is an aliasing problem (and it is not due to imprecision of the analysis), we checked all the places where this method is called. Also, notice the comment which states "*This should really only be called by Customer.call(..)*". We found all callers and verified that this comment is obeyed using the call graph visualization, by clicking on the SA icon next to the method signature. This pops up the two call graph links. The first link indicates that the only place where this constructor is invoked is in `Customer.call()`, confirming the comment. The second link indicates that this constructor implicitly invokes the constructor of `java.lang.Object`. By clicking the first link, we were taken to the call site in `Customer.call()`, a very short method containing little of interest. We followed another call graph link to the only call site calling `Customer.call()`, in `CallThread.run()`. By looking through the code of this method (see Figure 5(b)), we see that the caller and receiver are simply chosen at random from the list of customers, with no code to ensure that they are distinct. We have indeed found a bug, which we can observe at runtime in the output snippet in Figure 5(c), in which `Customer8` calls himself.

In general, we believe that by visualizing the call graph and aliasing computed by the points-to analysis in Soot, we can provide more accurate information to the end user in a way that enables them to browse and understand their program.

5.3 Unreachable Method Analysis

Another application of the call graph computed by Soot is flagging methods which are unreachable through the call graph from the starting point of the program. These

⁴ Note that this aliasing information was computed using a fairly precise interprocedural points-to analysis, which means that objects from the same allocation site may reach both of these parameters.

The screenshot shows an IDE interface with two main panes. The left pane displays Java code for a `Call` class. The right pane shows a call graph for the `Call` class, with nodes for `caller`, `receiver`, `connections`, and methods like `Call(Customer)`, `pickup()`, `isConnected()`, and `merge(Call)`. A red box highlights the parameter `Customer receiver` in the code, which corresponds to the `receiver` node in the call graph.

```

/*
package telecom;
import java.util.Vector;
import java.util.Enumeration;

/**
 * A call supports the process of a customer trying to
 * connect to others.
 */
public class Call {

    private Customer caller, receiver;
    public CallGraph Source Type: SPECIAL Source Method/Context: <telecom.Customer>; telecom.Call call(telecom.Customer) CallGraph Type: SPECIAL Target Method/Context: <java.lang.Object>; void <init>()
    /*
     * Create a new call connecting caller to receiver
     * with a new connection. This should really only be
     * called by Customer.call(..)
     */
    public Call(Customer caller, Customer receiver) {
        this.caller = caller;
        this.receiver = receiver;
        Connection c;
        if (receiver.localTo(caller)) {
            c = new Local(caller, receiver);
        } else {
            c = new LongDistance(caller, receiver);
        }
        connections.add(c);
    }
}

```

(a) aliased parameters and call graph links

```

boolean foundFreeCaller = false;
do {
    int index = Math.abs(r.nextInt())%(AbstractSimulation.customers.size());
    caller = (Customer) AbstractSimulation.customers.get(index);
    if (!caller.lineIsBusy())
        foundFreeCaller = true;
}while(!foundFreeCaller);

boolean foundFreeReceiver = false;
do {
    int index = Math.abs(r.nextInt())%(AbstractSimulation.customers.size());
    receiver = (Customer) AbstractSimulation.customers.get(index);
    if (!receiver.lineIsBusy())
        foundFreeReceiver = true;
}while(!foundFreeReceiver);

...
Call c1 = caller.call(receiver);

```

(b) code extract for creating a telephone call between a caller and receiver

```

CURRENT TIME 10:55
Thread 2 Start
Customer8 calls Customer8
[new local connection from Customer8(514) to Customer8(514)]
Call length 17
Customer8 accepts call
connection completed

```

(c) program output extract illustrating bug

Fig. 5. Example of aliased parameters and call graph links

methods could be removed to reduce the size of the program, either for performance reasons, or to make the program easier to understand. We highlight these methods in red using a Colour Tag, and also add a String Tag to indicate that the method is unreachable. By taking advantage of the Spark [13] points-to analysis framework, Soot can produce call graphs with different levels of precision, and this translates into different levels of precision in detecting unreachable methods. To demonstrate the effect of the precision gained from Spark, we describe two (out of many) examples of unreachable methods that were detected in benchmarks using a precise call graph built with Spark, but that cannot be determined to be unreachable using a simpler call graph construction algorithm such as Class Hierarchy Analysis (CHA) [9].

The Jess benchmark from the SPECjvm98 [20] suite contains a class called `LostDisplay`. The code using this class is commented out, with a comment indicating that it was commented out when the original Jess code was made into a SPEC benchmark. However, a field of type `LostDisplay` remains, so the class is not entirely unused. A different implementation of the same interface also exists (called `NullDisplay`) and its methods are called by the benchmark. Therefore, the methods of `LostDisplay` cannot be determined to be unreachable with a call graph based on CHA, but by using a more precise analysis such Rapid Type Analysis [2], Soot is able to determine all but one method of `LostDisplay` to be unreachable.

The SableCC [19] parser generator is another interesting example. Like many programs, it includes its own utility library with implementations of standard data structures such as lists and trees. As with most libraries, the application does not use all of the library code. However, because the data structures are organized in a class hierarchy as various implementations of general interfaces, and because the application is coded to the interfaces rather than the implementations, a CHA-based call graph is insufficient to determine which methods of which implementations are unused. By constructing the call graph with the help of a Spark points-to analysis, Soot can flag many more of these library methods as unreachable.

Thus, we can see that by exposing the more accurate analysis in Soot to the end user, we can improve the tools that aid in program understanding or refactoring.

5.4 Array Analysis

In this final example, we present an application of our framework to a more advanced analysis, more typical of a use in research. As part of our earlier research on array bounds check removal, we implemented in Soot a comprehensive array bounds analysis [18], consisting of three related analyses: *variable constraint analysis*, *array field analysis*, and *rectangular array analysis*. In that paper, we provided quantitative results about the effectiveness of these analyses on various benchmarks, in terms of the number of bounds checks eliminated and the speedup obtained. However, we are also interested in qualitative results of the analysis, for a number of reasons, from the point of view of both the programmer writing the benchmark, as well as the designer of the analysis.

For the programmer, it is very reassuring when an analysis determines that none of the array accesses in the program will throw an `ArrayIndexOutOfBoundsException`. It is even more important, however, for the analysis to provide sufficient feedback when it cannot prove that an array access is safe, so that the programmer can study the reasons

why the analysis was unsuccessful, and determine whether it is due to imprecision in the analysis, or an actual bug in the program. In addition, the programmer may be interested in the effect of specific optimizations on the performance of the program. If the programmer is relying on the compiler to perform some optimization, it is useful for the compiler to report whether it has enough information to perform the optimization, and if it does not, what would need to be changed for the optimization to be applicable.

While the array bounds check elimination was very effective on some benchmarks, it was less effective on others. As compiler researchers, we want to know why it failed on these other benchmarks, and what could be done to fix it. Like many compiler optimizations, the array bounds check elimination is dependent on the three interdependent analyses, which in turn depend on other information that the compiler has computed, such as the call graph. Therefore, even for the developer of the analyses, it is difficult to determine only by examining the benchmark code the results that the various analyses will produce, and whether the optimization will be applied. However, by visualizing the results of each analysis on the benchmark code, the analysis developer can quickly see which analyses are failing to deduce the results needed by dependent analyses and optimizations.

To visualize the results of the array bounds check elimination, we used four colours of Colour Tags to highlight the array index expression, to indicating one of the four possible combinations of the upper and lower bound being safe or unsafe. In addition, for multi-dimensional array expressions, we used one of two colours to indicate whether the array had been determined to be rectangular. Finally, when studying why certain arrays known to be rectangular were not found to be rectangular by the analysis, we found it convenient to also generate the call graph Link Tags, since the rectangular array analysis is interprocedural and dependent on the call graph. These visualizations were very helpful in determining why the analysis failed on certain benchmarks, and has helped us identify ways to improve the analysis.

Overall, we believe that appropriate visualizations will aid compiler researchers to quickly view the results of their analyses, to identify places where they expect the analysis to be more precise, and to help browse the program to identify where the analysis became imprecise.

6 Related Work

Several projects have attempted to visually display the data structures inside compilers for teaching and for debugging compiler optimizations. Xvpodb [7] displays the intermediate representation at various stages in a static optimizer. The Feedback Compiler [4] shows animations of the transformations performed on an intermediate representation in the 1cc [11] compiler. The Visual Simple-SUIF Compiler [12] displays the intermediate representation, as well as control flow graphs, data dependence graphs, and register interference graphs. Vegdahl [24] presents a visualizer intended primarily for teaching, which shows the data structures in the front-end, rather than back-end, of a compiler.

A number of tools in early stages of development present interprocedural information such as the call graph to Java programmers. These include JAnalyzer [6], a tool based on Soot for browsing call graphs, and the Gnosis framework [3], another call graph

construction toolkit. Chang, Jo, and Her [8] have developed a tool for visualizing the interprocedural control flow due to exceptions in Java programs.

More broadly, a large research community exists with entire conferences devoted to software visualization and program comprehension [10, 15]. Most of this work focuses on high-level visualization of the overall structure of large pieces of software, in contrast with our work, which focuses on showing information gained from analyzing the code statement by statement. On the other extreme, a large body of work exists on using program slicing [22, 5] to help programmers understand the effects of individual statements.

An interesting point of our work is that we are providing a toolkit which makes it easy to integrate the power of an optimizing compiler into an existing IDE. By implementing it as an Eclipse plugin, it can be widely used (anybody using Eclipse can easily add the plugin) and the resulting integrated tool is useful for both compiler developers and end users.

7 Conclusions and Future Work

We have integrated the Soot compiler infrastructure into the Eclipse IDE. We identified three major challenges to providing such an integrated toolkit and described how we tackled these challenges and described our approach for designing the Soot plugin to Eclipse. An important part of our solution is a generic visualization framework for conveying analysis results from the compiler to the user. This provides a rich environment for programmers seeking to better understand their code, students learning about compiler analyses, and compiler researchers studying new analyses. We have shown the significance of several visualizations already implemented in the framework, and we plan to implement other visualizations in the future.

Another potential area of future work is construction of a step-through analysis debugger. The idea is to develop a debugger-like interface that allows the user to step through a data flow analysis as it is being performed, and visualize the information being propagated. This would be a welcome tool both for students learning about data flow analyses, and for experienced researchers debugging complicated analyses.

Some of the output produced by Soot takes the form of graphs, such as control flow graphs and call graphs. Our current preferred method of drawing these graphs is by producing input to the dot graph drawing tool. It would be useful to integrate such a graph drawing tool into Eclipse, to make it possible to view these graphs in Eclipse. The graphs could even be animated (to step through a data flow analysis on a control flow graph, for example) or interactive.

References

1. AspectJ. <http://eclipse.org/aspectj/>.
2. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341. ACM Press, 1996.

3. I. Balaban, F. Tip, and J. Dolby. Using static analysis for program understanding in an integrated development environment <http://oopsla.acm.org/fp/files/dem-1.html>, 2002.
4. D. Binkley, B. Duncan, B. Jubb, and A. Wielgosz. The feedback compiler. In *IEEE Sixth International Workshop on Program Comprehension*, pages 198–205, June 1998.
5. D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.
6. E. Bodden. A High-level View of Java Applications. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications (Companion)*, SIGPLAN, Anaheim, CA, October 2003.
7. M. Boyd. Graphical visualization of compiler optimizations. Master’s thesis, Florida State University, July 1993.
8. B.-M. Chang, J.-W. Jo, and S. H. Her. Visualization of exception propagation for Java using static analysis. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM’02)*, page 173. IEEE Computer Society, 2002.
9. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECCOP’95—Object-Oriented Programming, 9th European Conference*, volume 952 of *LNCS*, pages 77–101, 7–11 Aug. 1995.
10. S. Diehl and J. T. Stasko, editors. *Proceedings of the 2003 ACM Symposium on Software Visualization*. ACM Press, 2003.
11. C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.
12. B. K. Harvey. Graphical user interface for compiler optimizations with Simple-SUIF. Master’s thesis, University of California, Riverside, December 1996.
13. O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, April 2003.
14. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
15. H. A. Müller, R. Koschke, and K. Wong, editors. *11th International Workshop on Program Comprehension (IWPC 2003)*. IEEE Computer Society, 2003.
16. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 138–152, April 2003.
17. P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Compiler Construction, 10th International Conference (CC 2001)*, volume 2027 of *LNCS*, pages 334–554, 2001.
18. F. Qian, L. Hendren, and C. Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *Compiler Construction, 11th International Conference*, volume 2304 of *LNCS*, pages 325–341, April 2002.
19. SableCC. <http://www.sable.mcgill.ca/sablecc/>.
20. SPEC JVM98 Benchmarks. <http://www.spec.org/osg/jvm98/>.
21. V. Sundaresan, L. Hendren, C. Razafimahefana, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of OOPSLA 2000*, pages 264–280, 2000.
22. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
23. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, volume 1781 of *LNCS*, pages 18–34, 2000.
24. S. R. Vegdahl. Using visualization tools to teach compiler design. *The Journal of Computing in Small Colleges*, 16(2):72–83, 2001.

Declarative Composition of Stack Frames

Christian Lindig¹ and Norman Ramsey²

¹ Universität des Saarlandes, 66123 Saarbrücken, Germany

lindig@cs.uni-sb.de

² Division of Engineering and Applied Sciences

Harvard University, Cambridge, MA 02138, USA

nr@eecs.harvard.edu

Abstract. When a compiler translates a procedure, it must lay out the procedure’s stack frame in a way that respects the calling convention. Many compilers respect the convention either by restricting the order in which slots can be allocated or by using different abstractions *ad hoc* for allocating in different regions of a frame. Such techniques cause the implementation of the layout to be spread over much of the compiler, making it difficult to maintain and verify. We have concentrated the implementation of layout into a single, unifying abstraction: the block. The block abstraction decouples layout from slot allocation. Stack-frame layout is specified in one central place, and even complex layouts are achieved by composing blocks using only two simple operators. Our implementation is used in the Quick C-- compiler to support multiple calling conventions on multiple architectures.

1 Introduction

In a compiled language, most of the information specific to one procedure is stored in that procedure’s *stack frame*. The stack frame may hold such information as local variables, spilled temporaries, saved registers, and so on. The stack frame can even hold a source-language record or object, provided the object does not outlive the activation of its procedure. The advantage of allocating so much information into the stack frame is that everything can be addressed using only one pointer: the frame pointer. Each object or private datum is addressed by adding a different *offset* to the frame pointer.

Allocating memory on a stack performs so well that it is used in almost all compilers,¹ but it is nevertheless tricky:

- The offsets of some objects depend on the sizes of other objects, but the compiler must generate code before the size of every object is known. For example, the compiler might have to generate a reference to a record allocated on the stack before knowing how much stack space will be needed to hold callee-saves registers.

¹ For a notable exception see Appel (1992).

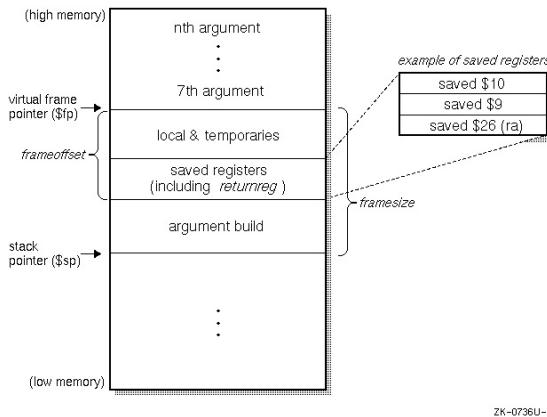


Fig. 1. Frame layout for the Tru64 Unix platform, from the online documentation of the *Assembly Language Programmer's Guide* (Compaq, 2000, Chapter 6).

- The compiler is not completely free to decide on the layout of a stack frame. To interoperate with other compilers or with debuggers, garbage collectors, and other tools that may walk the stack, the compiler must lay out the stack frame in a way that respects an architecture-specific calling convention (Compaq, 2000; Intel, 2003). Such a convention typically divides a frame into *regions*, each of which holds one kind of data: arguments or saved registers, for example.
- The compiler is not given the contents of all regions at once; instead, it must allocate stack space incrementally, one slot at a time. If the time at which a slot is allocated determines its position in the stack frame, the requirement to respect the calling convention places a heavy constraint on the order in which different parts of the compiler may execute.

The ideal compiler should be prepared to allocate any slot in any region at any time from any part of the compiler, all while making sure that the emerging frame layout respects the calling convention.

Today's compilers manage this problem, but at a cost in complexity. Compilers use different abstractions for different kinds of slots, and these abstractions are spread over the compiler. For example, the GCC compiler uses `temp_slot` values for temporaries and `arg_data` values for arguments, and these abstractions are defined in different modules (FSF, 2003). Similarly, the Objective Caml compiler OCAMLOPT uses different abstractions for slots for temporaries, incoming arguments, and outgoing arguments (Leroy et al., 2002).

The difficulty is understanding how these various abstractions interact as they form the frame. Frame layout is almost always specified by a diagram like that in Fig. 1, but in practice it is almost impossible to find the implementation of this diagram, because the implementation is spread over the compiler. Even given both the diagram and the compiler's source code, it can be quite difficult to verify that frame layout is correct—let alone change it. And such

changes can be useful; for example, by adding a “canary” in front of a saved return address, one can protect against buffer overruns (Cowan et al., 1998). Other tasks that are unnecessarily difficult include arguing that the frame is as small as possible or experimenting with alternative layouts for improved performance (Davidson and Whalley, 1991). These difficulties are multiplied when a compiler supports multiple back ends or multiple calling conventions.

To address these concerns, we have developed a unifying abstraction that supports incremental allocation of stack slots in any order. The abstraction is based on declarative, hierarchical composition of memory blocks. It is unifying in that a slot, a region, and even the frame itself can each be represented as a block. The layout of the stack frame is specified using two block-composition operators: overlapping and concatenation. With this abstraction we have achieved the following results:

- Frame layout is specified using an algebraic description that corresponds directly to the sorts of diagrams found in architecture manuals (e.g., Fig. 1). The specification appears in one central place in the compiler.
- As a bonus, the layout can even be specified at compile time, instead of the more typical compile-compile time. Such flexibility facilitates experiments with frame layouts.
- Block creation, addressing, and composition are decoupled. Blocks can be created and composed in any order and any part of the compiler, thus enabling the compiler writer to run the parts of the compiler in any order, independent of the frame layout.

Our block-composition abstraction is implemented in the Quick C++ compiler, a new compiler for the portable assembly language C++ (Ramsey and Peyton Jones, 2000). The compiler produces code for Alpha, IA-32, IA-64, MIPS, and Power PC architectures. Its back end is controlled by the embedded scripting language Lua (Ierusalimschy et al., 1996; Ramsey, 2004), which allows a user to define frame layout as part of a calling convention.

2 Background

The layout of a stack frame must meet the requirements of the procedure calling convention. This convention is actually a contract among *four* parties: a calling procedure (the *caller*), a called procedure (the *callee*), a run-time system, and an operating system. All four parties must agree on how space will be allocated from the stack and how that space will be used: A procedure needs stack space for saved registers and private data, an operating system needs stack space to hold machine state (“signal context”) when an interrupt occurs, and a run-time system needs to walk a stack to inspect and modify the state of a suspended computation. In addition to sharing the stack with the other parties, a caller and callee must agree on how to pass arguments and results. This paper focuses on the layout of a stack frame; a companion paper (Olinsky et al., 2004) addresses

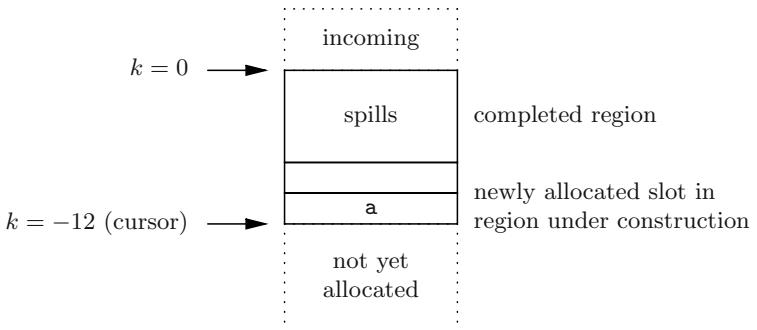


Fig. 2. Contiguous allocation of a stack frame using a cursor

the other big part of a calling convention: how procedures use registers to pass arguments and results.

For our purposes, then, the key fact about a calling convention is that it divides a stack frame into regions. A single region contains one kind of data, such as saved registers, arguments passed on the stack, source-language objects, and so on. A region is a sequence of *slots*, where each slot holds a single datum. The slot is the unit of allocation; for example, when the compiler discovers it needs to pass an argument on the stack, it allocates a slot to hold that argument.

At run time, a slot is read and written using an address of the form $p + k$, where p is typically either the stack pointer or the frame pointer, and k is a constant offset. A key question is when k is known. The answer is determined by the allocation strategy.

- One strategy is to allocate slots consecutively from the top of the frame to the bottom, as shown in Fig. 2. The compiler maintains a *cursor*, which keeps track of the amount of space allocated. To allocate a slot, the compiler advances the cursor by the size of the slot. Should the slot require some alignment, the compiler aligns the cursor beforehand.

When slots are allocated using a cursor, k is known as soon as a slot is allocated. Knowing k immediately makes it easy to emit instructions that refer to the slot, but this allocation technique couples frame layout to the execution of the compiler: Allocations must be executed in an order consistent with the calling convention. This restriction can lead to contortions; for example, the LCC compiler requires the same compiler phases to execute in different orders depending on the target architecture (Fraser and Hanson, 1995).

- The other strategy is to allocate slots from different regions independently. In this strategy, k cannot be known until the sizes of earlier regions are known, i.e., not until allocation is complete. Not knowing k means that in an instruction that refers to the slot, k must be represented by a symbolic address, which must be replaced later. But this allocation technique makes frame layout independent of the execution of the compiler: Slots can be allocated in any order.

We use the second strategy, which maximizes the compiler writer’s freedom. To make it easy to compute k , we introduce a new abstraction: the *block*.

3 Blocks and Block Composition

A block can represent a single slot, a region of the stack frame, a group of regions, or even the entire frame. Whatever it represents, a block is completely characterized by four properties:

- A *size* s , in bytes
- An *alignment* a , also in bytes
- A *base address* b , which is a symbolic expression
- A set of *constraints* c , where each constraint is a linear equation over base addresses

Size, alignment, and base address describe a region of memory. The base address is constrained by the alignment: $b \bmod a = 0$. After frame layout is determined, each base address will be equivalent to an addressing expression of the form $p + k$, e.g., $b = p + k_b$. The constraints express relationships, as described below, among base addresses of blocks that are in the same frame.

To form stack frames, blocks are composed in conventional ways, but ultimately convention is based on interference. Two blocks interfere if they are live at the same time, in which case they cannot share space. Interfering blocks are therefore *concatenated*. For example, source-language variables and spilled temporaries might be concatenated. If two blocks do not interfere, they are never live at the same time, in which case they can share space. Non-interfering blocks are therefore *overlapped*. For example, arguments passed on the stack at two different call sites might be overlapped. Both the concatenation and overlapping operations are designed so that a composed block contains its constituents, and correct alignment of the composed block implies correct alignment of the constituents. A composed block is as large as necessary, but no larger.

The ideas of concatenation and overlapping are simple, but some details are tricky. We therefore present both operations in detail, with formal semantics. In these presentations, we use two forms of notation. In short form, we write a block with size s , alignment a , and base address b as $saa@b$, pronounced “ s bytes aligned a at b .” For example, $12a4@data$ means a 16-byte block aligned on a 4-byte boundary at address `data`. We write constraints, if any, on the side. In long form, we use functions *size*, *base*, *align*, and *constr* to refer to the four properties of a block. The two forms are related by the equations $\text{size}(saa@b) = s$, $\text{align}(saa@b) = a$, and $\text{base}(saa@b) = b$.

As examples, we use blocks named after regions of a stack frame: block `in` holds incoming arguments, block `out` holds outgoing arguments, block `spills` holds spilled temporaries, block `conts` holds “continuations” (used to implement exceptions in C--), and block `data` holds user-allocated data.

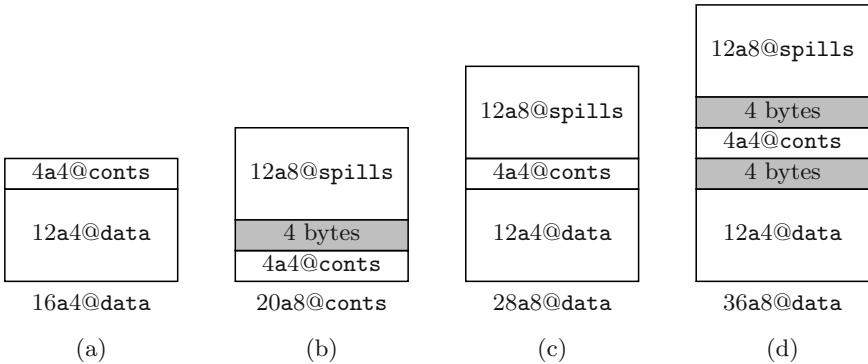


Fig. 3. Concatenation of memory blocks that form the private part of a frame. Notation **12a4@data** means a block of size 12, aligned on a 4-byte boundary, at address **data**; constraints are not shown. Padding (shaded) is required to preserve alignment. Padding is sensitive to order; for example, composition (d) contains the same blocks as composition (c), yet is much larger.

3.1 Concatenation

We write concatenation using the infix operator \oplus ; for example, we write the concatenation of **12a4@data** and **4a4@conts** as **12a4@data** \oplus **4a4@conts**, and the second block begins where the first block ends (Fig. 3a). The address of the concatenation is the address of the first block, so **12a4@data** \oplus **4a4@conts** is equivalent to **16a4@data** with the constraint **conts** = **data** + 12. Because the size of **12a4@data** is 12, a multiple of 4, base address **conts** is guaranteed to be 4-byte aligned provided address **data** is 4-byte aligned.

When concatenating blocks, we may need padding to guarantee the alignment of the right-hand block. For example, **4a4@conts** \oplus **12a8@spills** = **20a4@conts** with constraint **conts** + 8 = **spills**, where 4 bytes of padding are required (Fig. 3b). The padding makes **spills** – **conts** a multiple of 8.

As these examples show, proper alignment of a concatenation **lo** \oplus **hi** guarantees proper alignment of **lo** and **hi**. The guarantee comes from two decisions: the alignment of the concatenation is the least common multiple of the alignments of **lo** and **hi**, and the distance between the base of **lo** and the base of **hi** is a multiple of the alignment of **hi**. The concatenation is therefore defined by the following equations:

$$\begin{aligned}
 \text{base}(\text{lo} \oplus \text{hi}) &= \text{base}(\text{lo}) \\
 \text{size}(\text{lo} \oplus \text{hi}) &= \text{round_up}(\text{size}(\text{lo}), \text{align}(\text{hi})) + \text{size}(\text{hi}) \\
 \text{align}(\text{lo} \oplus \text{hi}) &= \text{lcm}(\text{align}(\text{lo}), \text{align}(\text{hi})) \\
 \text{constr}(\text{lo} \oplus \text{hi}) &= \text{constr}(\text{lo}) \cup \text{constr}(\text{hi}) \cup \\
 &\quad \{ \text{base}(\text{hi}) = \text{base}(\text{lo}) + \text{round_up}(\text{size}(\text{lo}), \text{align}(\text{hi})) \} \\
 \text{round_up}(x, n) &= n \times \lfloor (x + n - 1)/n \rfloor
 \end{aligned}$$

Concatenation has one awkward property: When three or more blocks are concatenated, padding can depend on the order of composition. In other words, concatenation is not associative. For example, columns c and d of Fig. 3 show two different compositions of the blocks $12a4@\text{data}$, $4a4@\text{conts}$, and $12a8@\text{spills}$. The composition $(12a4@\text{data} \oplus 4a4@\text{conts}) \oplus 12a8@\text{spills} = 28a8@\text{data}$, with constraints $\text{data} + 12 = \text{conts}$ and $\text{data} + 16 = \text{spills}$, and the composition $12a4@\text{data} \oplus (4a4@\text{conts} \oplus 12a8@\text{spills}) = 36a8@\text{data}$, with constraints $\text{conts} + 8 = \text{spills}$ and $\text{data} + 16 = \text{conts}$. The first composition needs no padding, but the second needs 8 bytes. The reason is that the second composition uses a concatenation on the right: The alignment of the right-hand block determines the padding, and a composite block has an alignment constraint at least as large as the constraints of its constituents.

Luckily, the nonassociativity of concatenation presents no problems in practice. In our compiler, we specify frame layout using an operation that concatenates a *list* of blocks. This operation applies the binary operator \oplus in a left-associative way, which minimizes padding. Associativity is not an issue in cursor-based stack allocation because there is no choice of order; padding is added on demand for each slot, and the results are equivalent to our implementation.

3.2 Overlapping

When two blocks are never live at the same time, they can overlap, sharing memory. When two overlapping blocks are of different sizes, it may be possible to place the smaller block anywhere within the larger block, but in practice, the smaller block is placed at the bottom of the larger block, so that the two base addresses are equal. We call this operation *overlap low*. For completeness, we also discuss the *overlap high* operation, in which the smaller block is placed at the top of the larger block.

The most common use of the overlap-low operation is to create an argument-build area, which holds arguments that are to be passed to a callee on the stack. The area must be large enough to hold the arguments for the most complicated call in the procedure's body. Such an area can be created by allocating each call site's arguments into a different block, then overlapping all such blocks.

For example, on the Tru64 Unix platform the first six arguments are passed in registers and later arguments on the stack. Let us assume a procedure body with two call sites, where the first passes one argument on the stack, and the second passes two arguments on the stack. Each argument is 64 bits, i.e., 8 bytes. According to the ABI (Compaq, 2000), a region holding arguments must be 16-byte aligned. Therefore, the first call reserves a block $8a16@\text{out}_1$, and the second a block $16a16@\text{out}_2$. When the compiler later assembles the frame, it overlaps these blocks at their low end and arrives at a $16a16@\text{out}_1$ block and constraint $\text{out}_1 = \text{out}_2$.

Overlapping two blocks at their low ends is easy. The result is large enough to hold either block and is aligned for both. The constraint specifies that both blocks have the same base address.

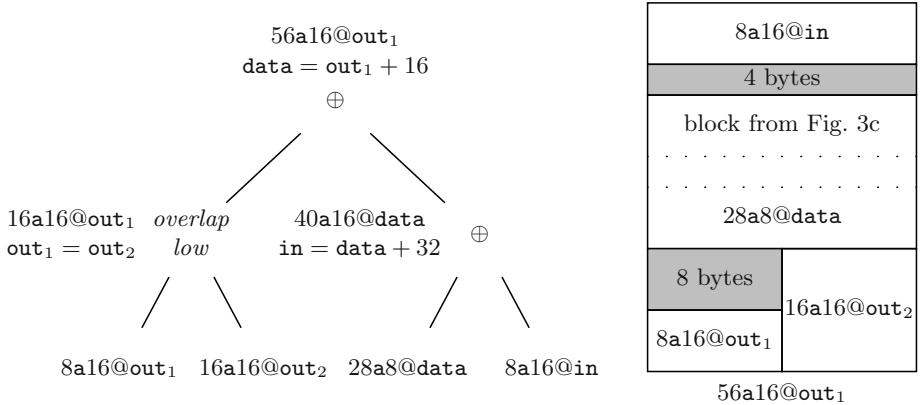


Fig. 4. Composition of four blocks into a frame: incoming arguments, private data (from Fig. 3c), and two blocks for outgoing arguments. The left side shows the tree-shaped construction, the right side the resulting block layout. Concatenation is denoted by \oplus , overlapping at the low end by $overlap_low$. Blocks $16a16@out_1$ and $16a16@out_2$ overlap such that they align on their lower end (higher addresses are towards the top of the diagram). To guarantee the alignment of in , padding was added.

$$\begin{aligned}
 base(overlap_low(x, y)) &= base(x) \\
 size(overlap_low(x, y)) &= max(size(x), size(y)) \\
 align(overlap_low(x, y)) &= lcm(align(x), align(y)) \\
 constr(overlap_low(x, y)) &= constr(x) \cup constr(y) \cup \{base(x) = base(y)\}
 \end{aligned}$$

Surprisingly, overlapping blocks at their high ends is possible only if the difference between the blocks' sizes is a multiple of the smaller block's alignment. Otherwise, the smaller block cannot be placed at the high end of the larger block while simultaneously maintaining proper alignment of both base addresses. Padding cannot help: Padding the smaller block at the top would move it away from the high end of the larger block, and padding the larger block at its base would destroy its base alignment. The $overlap_high$ operator is therefore defined only when its precondition is satisfied.

$$\begin{aligned}
 overlap_high(x, y) &= \perp \text{ if } (size(y) - size(x)) \bmod align(x) \neq 0 \\
 base(overlap_high(x, y)) &= base(y) \text{ if } size(x) \leq size(y) \\
 base(overlap_high(x, y)) &= base(x) \text{ if } size(x) > size(y) \\
 size(overlap_high(x, y)) &= max(size(x), size(y)) \\
 align(overlap_high(x, y)) &= lcm(align(x), align(y)) \\
 constr(overlap_high(x, y)) &= constr(x) \cup constr(y) \cup \\
 &\quad \{base(x) + size(x) = base(y) + size(y)\}
 \end{aligned}$$

The precondition on $overlap_high$ depends on both blocks, and the only easy way to ensure that $overlap_high$ is possible is to pick a basic alignment (such

```

⟨Lua compiler configuration⟩≡
function Alpha.layout["C"](dummy, proc)
    local blocks = Stack.blocks(proc)
    blocks.in    = Block.overlap_low(64, blocks.youngblocks)
    blocks.out   = Block.overlap_low(64, blocks.oldblocks)
    local layout =                      -- <-- high memory
        { blocks.in           -- incoming arguments
        , blocks.vfp          -- virtual frame pointer
        , blocks.spills        -- temporaries
        , blocks.conts         -- continuations (for exceptions)
        , blocks.data          -- user-allocated data
        , blocks.sp            -- stack pointer
        , blocks.out           -- argument build
    }                           -- <-- low memory
    local block = Block.cat(64, layout)
    block = Block.adjust(block)  -- round up size to multiple of 16
    Stack.freeze(proc, block)   -- resolve addresses
    return 1
end

```

Fig. 5. Lua procedure, from Quick C--, that defines the frame layout for the Tru64 Unix platform. Compare with Fig. 1.

as word size) for all blocks and to make sure every block’s size is a multiple of that alignment. This restriction may explain why *overlap_high* is rarely used in conventional frame layouts.

4 Specification of Frame Layout

The block-composition operators described above suffice to create a specification of frame layout that not only can be executed in the compiler but also can be compared with a diagram found in an architecture-specific ABI manual. In the Quick C-- compiler, this specification takes the form of a function written in the embedded scripting language Lua (Ierusalimschy et al., 1996; Ramsey, 2004), which does for Quick C-- what ELisp does for Emacs. As an example, Fig. 5 shows Lua procedure `Alpha.layout`, which composes the frame for the C calling convention on the Alpha Tru64 Unix platform. This procedure is executed each time the compiler lays out a stack frame. It lays out the frame using the table `blocks`, which contains all the blocks for the procedure `proc` under translation. The procedure begins composing the frame by overlapping all the blocks that can share space. For example, list `blocks.youngblocks` holds a block for each call site; each block reserves space for outgoing arguments passed on the stack. `Alpha.layout` overlaps them at their low end to form the argument-build area `blocks.out`. Similarly, `blocks.oldblocks` holds blocks for incoming arguments and any results that are returned on the stack; these blocks are overlapped to form `blocks.in`. The procedure then forms the list `layout`, which contains the

blocks that represent the regions of the stack frame, in the order in which they should be laid out. Regions `blocks.vfp` and `blocks.sp` are empty blocks whose base addresses correspond to the virtual frame pointer and the stack pointer, respectively. The concatenation is done by `Block.cat`, which repeatedly applies concatenation operator \oplus , associating to the left.

Comparing Fig. 5 with the official specification in Fig. 1 reveals that the correspondence between our implementation and the manual is close but not exact. Our `blocks.in` and `blocks.out` correspond exactly to the incoming arguments and argument-build area of Fig. 1. Similarly, our `blocks.vfp` and `blocks.fp` correspond exactly to the pointers in Fig. 1. But our treatment of locals, temporaries, and saved registers is different. Because our compiler works with a nonstandard run-time system, we do not need to distinguish saved registers from spilled temporaries. And because of the way our compiler does allocation, we find it convenient to distinguish two kinds of locals: `blocks.data` and `blocks.conts`. These kinds of detailed differences would be very difficult to extract from a conventional implementation of frame layout. It is the ability to compare the implementation directly with a picture that is missing in other compilers. As a bonus, blocks make it trivial to change frame layout.

5 Implementation

An implementation can use blocks in many ways: one block per region, one block per slot, or even one block per slot in some regions and one block per region in other regions. Which arrangement is best depends on whether it is necessary to control the order of slots in a region; for example, it might be desirable to control the order in which callee-saves registers are stored. Different arrangements require slight variations, but for concreteness we assume one block per region, which is the arrangement used in our implementation.

Given one block per region, our compiler follows these steps:

1. For each region of the stack frame, a unique symbolic address is created.
2. Slots are allocated from various regions, in any order. Each region is given a mutable data structure that remembers alignment and uses an internal cursor to allocate from that region only. A slot's address is a function of the region's cursor and symbolic address. The address is then used to generate intermediate code.
3. When allocation is complete, each region (or each slot) is identified with a fresh, unconstrained block. The block is given the symbolic address of its region. The block's size and alignment come from the region's internal cursor and remembered alignment, and the block has no constraints.
4. If a stack pointer or frame pointer points to a particular location in the frame, an empty block is created to represent that location.
5. Using concatenation and overlapping operators, blocks are composed to form a single block, which represents the frame. This step is shown in Fig. 5.
6. The frame's constraints are solved, and the solution provides the value of each symbolic address. If any block composition was illegal, the problem shows up here as unsolvable constraints.

7. The value of each symbolic address is substituted in the intermediate code.
This step and the previous step are performed in Fig. 5 by `Stack.freeze`.

Most of the implementation details are straightforward; for example, the implementation of blocks follows directly from the equations in Sect. 3, and it takes less than 100 lines of code. But a few details are worth discussing here.

Our compiler bases stack addresses on a virtual frame pointer, written `vfp`, so when we create a region, we know its address has the form `vfp + k`, but we don't know k . We therefore give a new block the symbolic address `vfp + v`, where v is a fresh *late compile-time constant*. A late compile-time constant is an integer whose value is not known until late in compilation. When created, it is represented by a string, and when the frame layout is known, that string is replaced by the integer for which it stands.

There is one exception to the rule that each block has a symbolic address of the form `vfp + v`. The empty block that represents the location of the virtual frame pointer has the symbolic address `vfp`. This block appears in Fig. 5 as `blocks.vfp`.

When n blocks are composed to form the stack frame, it requires $n - 1$ block-composition operations. As shown in Sect. 3, each operation adds one equation to the set of constraints on a block. And because one block has the symbolic address `vfp` and each of the other $n - 1$ blocks has a symbolic address of the form `vfp + v`, the `vfps` cancel and we wind up with a system of $n - 1$ equations in $n - 1$ unknowns. These equations are solved using an iterative algorithm similar to those presented by Knuth (1986§585) and Ramsey (1996). Experiments show that this algorithm is efficient in practice.

We have considered two extensions to the simple picture presented above. In the first extension, a stack pointer is used as well as a frame pointer, and some blocks addresses have the form `sp + k` while others have the form `fp + k`. If the stack pointer moves, perhaps because the compiler supports `alloca()`, this arrangement amounts to solving two independent systems of equations. But if the stack pointer does not move, so the frame has a fixed size, this arrangement requires an extra equation `fp - sp = vframe`, where v_{frame} is a late compile-time constant that stands for the frame size. When the equations are solved, the solution for v_{frame} gives the actual frame size.

The other extension we have considered is to minimize frame size by using register-allocation techniques to manage slots. In this extension, each slot gets its own block, and the compiler must compute an interference graph for these blocks. It can then color the graph and overlap blocks that share the same color.

6 When Things Go Wrong

Not every block composition yields a useful frame layout. Potential errors include

1. Concatenating a block with itself, directly or indirectly, which may produce an unsolvable constraint on the address of that block

2. Providing no block with a known address, so there are more unknowns than equations
3. Providing two blocks with known, inconsistent addresses, so there are more equations than unknowns
4. Using a single late compile-time constant in multiple blocks, so there are more equations than unknowns

Each of these errors manifests as failure in the equation solver, and it can be hard to track such an error to its source. Because solvability is a global property of the whole composition, it is hard to detect violations early. Our most important debugging aid is a routine that prints blocks and their equations. Such printouts have helped us debug frame layouts quickly. It also helps to choose good names for late compile-time constants.

Errors are detected at compile time, when the compiler actually lays out a frame for a procedure. Frame layout is therefore *not guaranteed* to be successful. But if the compiler lays out one frame successfully, it is *extremely likely* to lay out all frames successfully. The reason is that if a composition is unsolvable in general, it is unsolvable for almost all block sizes. (The “almost” is because certain incorrect compositions, such as concatenating a block with itself, may be solvable for one block size, such as zero.) To effectively eliminate the possibility of a compile-time error in frame layout, then, one could test layout with random block sizes. In practice, such testing appears to be unnecessary.

7 Related Work

Other compilers lay out stack frames using either a cursor or a mix of heterogenous abstractions. For example, the LCC C compiler uses a cursor (Fraser and Hanson, 1995). It allocates memory linearly, so the offset of each slot is known when the slot is allocated. When LCC leaves a local scope or a call site, it rolls back the cursor so the relevant slots can be reused. Therefore, when LCC compiles a function with multiple call sites, the slots used to pass arguments share stack space: The “argument-build area” is large enough to hold arguments passed at any call site. LCC tracks the size by using a “high-water mark” to hold the maximum value of the cursor. LCC’s correctness depends on executing allocations and rollbacks in exactly the right order. By contrast, blocks make correct layout independent of the order of allocation.

The GNU C compiler (GCC) defines virtual hardware registers that address arguments and private data in the frame; these registers are replaced late in compilation by a constant offset from either the frame pointer or the stack pointer, depending on the target (FSF, 2003). When a virtual register points to private data, that space is represented by a type `temp_slot` defined in `function.c`. Values of type `temp_slot` can be allocated and freed; adjacent freed slots are combined to avoid fragmentation. When a virtual register points to arguments, that space is allocated using a cursor implemented in `calls.c`. Module `calls.c` also maintains a bitmap for allocated memory and high/low marks to implement overlapping similar to LCC.

GCC’s virtual hardware registers and their substitution achieve the same effect as late compile-time constants: They free the compiler from order constraints imposed by the frame layout. Joining adjacent `temp_slot` values resembles concatenation in our representation. But it is only defined for a part of the frame that holds temporaries; other parts are maintained by a different module with different mechanisms. By using a single, uniform representation (blocks), we make it possible to define and change frame layout in one central place (Fig. 5).

8 Conclusions

A compiler should not be required to allocate stack slots in the linear order demanded by a frame layout. For laying out an entire stack frame, therefore, using a cursor is inferior to using blocks, which enable a compiler to allocate stack slots in any order. But when order of slots within a region does not matter, it is very convenient to use a cursor in each region.

Defining the layout of a block by composing blocks with `cat` and `overlap` is declarative—it requires the user only to define the size and alignment of simple blocks. The layout of any intermediate block is automatically deduced. The block abstraction is conceptually close to layout specifications and diagrams as they used in architecture manuals (Compaq, 2000, Chapter 6; Intel, 2003, Chapter 6), and block layouts can be centralized *and* lifted from the compiler’s innards to its surface, as shown in Fig. 5. Giving a user control over the layout of frames makes it easier to do experiments such as specializing calling conventions for optimized performance (Davidson and Whalley, 1991) or changing stack layout to combat buffer overruns that overwrite return addresses (Cowan et al., 1998).

Overlapping blocks at their high end is a dubious concept. It appears to be a natural dual to overlapping at the low end, but it is not possible in general. Small wonder that typical C calling conventions, such as IA-32, IA-64, MIPS, PowerPC, and Tru64 Unix, use overlap-low exclusively. Overlap-low is a natural fit for conventions in which the stack grows downward and later arguments are passed at higher addresses.

Blocks perform as well as other approaches. Stack layouts are, if not the same, equivalent in cost to layouts produced by other approaches. Compile-time costs are negligible. Constraint solving may sound scary, but the constraints are simply linear equations; indeed, a compiler that optimizes array access may already contain a suitable solver.

The only potential drawback of blocks is in debugging: It can be awkward to debug a bad system of equations instead of debugging frame layout directly. But the awkwardness fades after a little experience, and with the aid of the tools described in Sect. 6 we have had little difficulty debugging frame layouts. And we have no reason to believe that distributed allocation, with layout dependent on order, is any easier to debug.

In a retargetable compiler, which requires slight variations in layout for different platforms, blocks have made it easy for us to visualize, understand, and implement these variations. Because layout is centralized, it is easy to implement

a new layout using the traditional method of copying and modifying an existing one. Overall, using blocks has made it significantly easier to deal with calling conventions.

Acknowledgements. This work has been supported by NSF grants CCR-0096069 and ITR-0325460 and by a gift from Microsoft Research. Lex Stein, Stephan Neuhaus, Wolfram Amme, Andreas Rossberg, Andreas Zeller, and the 319 writing group provided helpful comments on drafts of the manuscript.

References

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

Compaq. *Tru64 Unix – Assembly Language Programmer’s Guide*. Compaq Computer Corporation, 2000.

Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 63–78, Berkeley, 1998. Usenix Association.

Jack W. Davidson and David B. Whalley. Methods for saving and restoring register values across function calls. *Software — Practice and Experience*, 21(2):149–165, 1991.

Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

FSF. *GCC Internals Manual*. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA, 2003. Corresponds to GCC 3.4.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software — Practice and Experience*, 26(6): 635–652, 1996.

Intel. *IA-32 Intel Architecture Software Developers’s Manual, Vol. 1*. Intel Corporation., P.O. Box 7641, Mt. Prospect, IL 60056, 2003.

Donald Ervin Knuth. METAFONT: *The program*, volume D of *Computers & Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System 3.06: Documentation and User’s Manual*. INRIA, 2002. Available from <http://caml.inria.fr>.

Reuben Olinsky, Christian Lindig, and Norman Ramsey. Staged allocation: Engineering the specification and implementation of procedure calling conventions. Technical Report TR-02-04, Division of Engineering and Applied Sciences, Harvard University, 2004.

Norman Ramsey. A simple solver for linear equations containing nonlinear operators. *Software — Practice and Experience*, 26(4):467–487, 1996.

Norman Ramsey. Embedding an interpreted language using higher-order functions and types. *Journal of Functional Programming*, 2004. To appear. Preliminary version appeared on pages 6–14 of *Proceedings of the ACM Workshop on Interpreters, Virtual Machines, and Emulators*, June 2003.

Norman Ramsey and Simon L. Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 35(5):285–298, 2000.

Author Index

Balakrishnan, Gogul 5
Burrill, James H. 24

Cahoon, Brendon 24
Cai, Qiong 150
Chatterjee, Siddhartha 104
Chen, Tong 57
Chowdhury, Rezaul A. 24
Clauss, Philippe 120

Dai, Xiaoru 57
Djeu, Peter 24

Economopoulos, Giorgios 89

Franchetti, Franz 217

Gao, Lin 150
Gregg, David 250

Hall, Mary 185
Hatcliff, John 39
Hendren, Laurie 281
Hosking, Antony L. 167
Hsu, Wei-Chung 57

Johnson, Neil 265
Johnstone, Adrian 89

Kral, Stefan 217

Lhoták, Jennifer 281
Lhoták, Ondřej 281
Lin, Jin 57
Lindig, Christian 298
Lorenz, Juergen 217

McKinley, Kathryn S. 24
McPeak, Scott 73
Meister, Benoît 134
Mycroft, Alan 265

Necula, George C. 73
Nisbet, Andy P. 250

Özer, Emre 250
Owen, Tim 202

Parker, Erin 104

Ramsey, Norman 232, 298
Ranganath, Venkatesch Prasad 39
Redwine, Kevin 232
Reps, Thomas 5

Scott, Elizabeth 89
So, Byoungro 185
Soffa, Mary Lou 1

Tchoupaeva, Irina 120

Ueberhuber, Christoph W. 217

VanDrunen, Thomas 167

Watson, Des 202
Wurzinger, Peter 217

Xue, Jingling 150

Yew, Pen-Chung 57